University of Hamburg, Department of Informatics

Bachelor Thesis

Flexible Event Imitation Engine

for Parallel Workloads

Jakob Lüttgau
Matrikelnummer 6146922
Bsc Informatik

18.03.2014

Supervisor Dr. Julian Kunkel
Second assessor Prof. Dr.-Ing. Stephan Olbrich

Evaluating systems and optimizing applications in high-performance computing (HPC) is a tedious task. Trace files, which are already commonly used to analyse and tune applications, also serve as a good approximation to reproduce workloads of scientific applications. The thesis presents design considerations and discusses a prototype implementation for a flexible tool to mimic the behavior of parallel applications by replaying trace files. In the end it is shown that a plugin based replay engine is able to replay parallel workloads that use MPI and POSIX I/O. It is further demonstrated how automatic trace manipulation in combination with the replay engine allows to be used as a virtual lab.

Contents

1. Introduction **4**
1.1. Motivation 4
1.2. Replaying Traces 5
1.3. Related Work 6
1.3.1. SIOX 7
1.3.2. strace/ltrace 7
1.3.3. ioreplay 7
1.3.4. ReplayFS 7
1.3.5. TraceReplay 8
1.3.6. Parabench 8
1.3.7. DIOS 8
1.3.8. Comparison 9
1.4. Goals of the Thesis, Requirements and Contribution 11

2. Design **13**
2.1. Foundation for Flexible Replay 13
2.2. Activity Data Type 14
2.3. Minimizing Distortions 14
2.4. Replayability, Environment and State Management 15
2.4.1. Replayability 15
2.4.2. Environment Pre-Creation 15
2.4.3. State Management during Playback 16
2.5. Trace Manipulation 16
2.6. Easing Plugin Development 16
2.7. Virtual Lab 17

3. Implementation **19**
3.1. Implementation Choices 19
3.2. Activity Data type 19
3.3. Feign: Overview of all Components 20
3.4. Feign API and Internals 21
3.4.1. Plugin registration 21
3.4.2. Layer registration 21
3.4.3. The activity work flow 22
3.4.4. Timing 23
3.4.5. Plugin Helpers 24

3.5. Developing Plugins 24
3.6. Supporting POSIX I/O 25
3.7. Supporting MPI 25
3.8. Supporting SIOX 25

4. Evaluation 27
4.1. Overhead 27
4.1.1. Test setup 27
4.1.2. Results 28
4.1.3. Test System 28
4.2. Accuracy of the Replayed Trace 29
4.3. MPIOM 29
4.4. Using feign as a Virtual Lab 30
4.4.1. Dummy application 31
4.5. Comparing feign to the Related Work 32

5. Summary and Future Work 34
5.1. Summary 34
5.2. Future Work 34

A. Source Code 36

1. Introduction

This chapter gives a short introduction into the field of high-performance computing and highlights some of the problems with evaluating systems. It is then explained how replaying trace files can help to overcome many of these problems. After presenting related work, the goals of the thesis are outlined.

1.1. Motivation

High-performance computing (HPC) relies on incredible complex systems. Most of todays fastest computers listed in the TOP500[1] employ the concept of cluster computing. A cluster computer combines the computational power of many smaller computers (nodes) by connecting them through a network. Every node then solves a small portion of a bigger problem.
Cluster computers are complex by nature. In practice many different kinds of technologies and many components are used. Thousands if not millions of compute nodes share specialised network and storage systems, and new technologies are constantly emerging e.g. graphical compute units (GPU). However, especially storage and network technologies are not as fast advancing as processors did. In general we can produce data much faster than modern storage solutions can permanently store them. Thus introducing a potential bottleneck.
To compensate for this, storage systems themselves turned to massively parallel approaches forming their own complex subsystems. Data is distributed among thousands of hard drives to ensure data safety and file access at high speeds. A similar trend can be observed for networks. To achieve high *bisection bandwidth*[2] many different technologies are combined. As a fully connected topology is usually not economical, networks are organized to reflect common communication patterns using specialized network hardware (e.g. switches). Further more, heterogeneous approaches may be used where critical sections are connected using fiber while other sections communicate using the cheaper Ethernet.
As alternative systems emerge, users would like to know if and how their applications would benefit. Performance of scientific applications depends on the interplay of different hardware components. An application being efficient on one system does not necessarily guarantee efficiency on another system. Most notably are the discrepancies in utilizing the different system resources that characterize applications. Simple classifications based solely on computational and Input/Output (I/O) load requirements proved insufficient.

[1] `http://www.top500.org/` - Ranking of some of the worlds most powerful general purpose computers.
[2] The bandwidth between the parts of a network when split into two equal parts.

Instead seven and later thirteen archetypes, or "dwarfs" [Asanovic et al., 2006], have been identified that capture the algorithmic nature as well as the patterns of computation and communication.
These different levels of complexity introduce the need for highly sophisticated tools, to provide benchmarking, stress testing, debugging and forensic capabilities [Joukov et al., 2005] to operators and users of such systems.
It is common to use so called synthetic benchmarks to evaluate system performance. But they are usually designed to determine some kind of peak performance of a component. For example the TOP500 ranking is solely influenced [TOP, 2014] by the results of the well known LINPACK benchmark which is used to determine the maximum number of floating point operations per second (FLOPS).
The results of synthetic benchmarks are often only of little use to prospect the performance and the behavior of a system because of the mentioned characteristics of real application workloads. Given the cost of HPC systems and tight budgets, there is little room for mismanagement so it makes a lot of sense for operators, institutions and organisations alike to elaborate before a purchase.
While it is common to benchmark for applications on reference systems, there are some practical limitations. It can require considerable effort to get an actual application to run on another system in an efficient fashion. New optimisations such as compiler flags or parameter changes to the message passing interface (MPI) or I/O need to be applied. Also code modifications maybe necessary e.g. adding or retracting read-ahead advice. Adapting the whole application to evaluate the impact of alternative optimizations to avoid suffering from performance regressions often may be too time consuming.
Another aspect to consider is that calculations on HPC systems may be confidential, as it might be the case in industry, intelligence or military to protect trade or state secrets. Performance problems in I/O or communication often require to seek support from vendors or the open source community. Code that isolates the problem into a small test case can be hard to come up with, and providing the actual application again may not be an option.

1.2. Replaying Traces

One solution to many of these problems maybe to record the events or activity (e.g. file or network I/O) an application triggers and conserve it for later use and analysis in so called trace files. Traces are already commonly used for debugging or post-mortem performance analysis of parallel applications. And many projects [PLF, 2014, **?**, Vam, 2014] publish trace files for verification or to engage other researchers.

```
time, activity1, param1, param2
time, activity2, param1, param2, param3
time, activity1, param1, param2
...
```

Figure 1.1.: A trace file typically holds information about the "order of" and "details about" the activities recorded.

Being able to replay activity recorded in trace files might make testing for performance of systems less work intense. A trace captures the characteristics of an application on the system it is run on, and may still be a good approximation on other systems. But many dependencies do not need to be satisfied. In fact sequential and some parallel benchmarks already use trace files. The advantage is that once implemented, replaying is a automated task, that can handle many different inputs.
As the level of detail in trace files is variable to some extend, confidential information are usually easy to remove. For example stress on a file system does not depend on the actual content but only on the size, the time and the type of an activity. But there may be exceptions e.g. the hierarchical data format HDF5 which is quite common in scientific applications, requires knowledge of the datafile for meaningful playback. An additional helper tool is needed to obfuscate the data while retaining the file structure.
Traces can be altered to reflect new features and remove old optimisation. A sophisticated replay environment that allows to systematically apply changes to a trace could then also serve as a virtual lab validating possible optimisations. Such would be very valuable for autonomous optimisation tools where machine learning approaches can utilize the replay environment to gain insights on isolated trace fragments to base their decisions on.
While there exist many solutions to obtain traces of parallel applications, there is only a handful of tools that offer to replay parallel workloads.

1.3. Related Work

To document the state of the art and to gain further insights on the demands users of replaying software have, we will take a look at related tools and projects. In Section 1.3.8 the features of the tools can be compared directly.
While we focus on parallel replay software, it still makes sense to also look into what has been achieved in non-parallel solutions.
Obtaining the trace data is not within the scope and responsibilities of this thesis, it is however important to realize that the trace strategy is a determining factor for the resulting imitation (e.g. driver level tracing, does not provide a full representation of file access patterns, because it does not account for the caches but rather can only provide insights on the penetration of the storage hardware). Consequently we start by looking at two solutions to trace applications.

1.3.1. SIOX

Scalable I/O for Extreme Performance (SIOX) [Kunkel et al., 2014] is a project that couples automatic monitoring and optimization of parallel I/O. The project is developed by members of the University of Hamburg, the ZIH Dresden and the HLRS Stuttgart.
Besides monitoring and optimizing parallel I/O, SIOX can be used as an extensive trace environment. A semi-automatic approach allows to support arbitrary layers by generating the instrumentation code from annotated header files. Activity captured with SIOX can then be permanently stored as a trace file. A library to read the trace is available. SIOX could integrate the results of a virtual lab as described in Sections 1.2 and 1.4 into its decision making process to issue optimisations.

1.3.2. strace/ltrace

Strace is a popular system call tracer that comes with most modern Linux and Unix distributions. Ltrace is the equivalent for library calls. Both are originally developed for sequential programs. The Los Alamos National Laboratory created LANL-trace [LAN, 2014], which is an application that builds upon these two tools, to gain parallel tracing capabilities. It determines skew and offsets of multi-node systems by running MPI before and after executing strace/ltrace.

1.3.3. ioreplay

Ioreplay [ior, 2014] is a tool to replay POSIX I/O, it is only working on Linux systems. It is relatively feature rich, but can currently replay only 20 systems calls. It can check in advance if the replay will succeed or abort with an error, but will not precreate the enviroment. It is possible to filter by call or by filename, and filenames can be mapped to another file. Further more traces can be converted into a binary format to minimize overhead. Ioreplay is part of a larger collection of tools called ioapps. It is implemented in C and open source.

1.3.4. ReplayFS

ReplayFS [Joukov et al., 2005], is an (non-parallel) replay application, that puts a focus on the forensic and debugging capabilities. It goes great lengths to achieve negative overhead, which lets you for instance replay a trace faster than recorded from the original application.
To do so, the developers had to move out of userspace into kernel space, to reduce any unnecessary layers and by that reducing the call-stack, thus saving time. They implemented their own specialised virtual file system (VFS) as a kernel module that would issue the file system operations. It can also restore the cache state. Be A pre procession run of is done before replay to minimize distortions.

1.3.5. TraceReplay

TraceReplay [Ahlers, 2012] is the result of a bachelor thesis, that successfully implements a prototype to show that a plugin based replayer is a viable concept with acceptable overheads. It comes with plugins to replay the most important POSIX operations and it is shown that plugin generation can be automated to some extend by adding annotations to the function signatures of header files.

1.3.6. Parabench

Parabench [Mordvinova et al., 2010] is a programmable parallel benchmark for POSIX. Benchmarks can be programmed and stored in a special script language. It allows to set up MPI groups and to generate reports from the measurements.

1.3.7. DIOS

DIOS [Kluge, 2011] is a parallel file system benchmark developed for during a PhD thesis at the TU Dresden, ZIH. Instructions are read from XML input, which also allow to demand parameter randomization. POSIX replay is realized by converting Vampirs [Vam, 2014] open trace format (OTF) into XML. It supports MPI parallel tasks aswell as MPI-I/O and POSIX I/O. It has not officially been released yet.

1.3.8. Comparison

To simplify the comparison of the different tools Figure 1.2 presents a comparison chart with the most important replay related features.

	ioreplay	ReplayFS	TraceReplay	Parabench	DIOS
Input	strace	tracefs	Plugins	PBL Script	XML, Vampir
Preprocessing	Yes	Yes	-	-	-
Binary Format	Yes	Yes	-	-	-
Sanity Check	Yes	-	-	-	Yes
Precreation	-	-	-	-	Pool[1]
Extandable	-	-	Plugins	Interpreter	-
Modifiers	Filepaths[2]	-	Filters	-	Expressions
offline	-	-	-	-	-
online	-	-	Yes	-	Yes
POSIX	Partial	Yes	Partial	Yes	Yes
Parallel	-	-	-	MPI	MPI
Threads	-	-	-	-	-
MPI	-	-	-	Yes	Yes
IO	-	-	-	Yes	Yes
Comm	-	-	-	Barrier,Groups	Barrier
Userpace	Yes	-	Yes	Yes	Yes
Kernel-module	-	Yes	-	-	-
Open Source	Yes	Yes	-	Yes	-
Dependencies	none	-	Glib	Glib, Bison	Bison, libxml
Prog. Language	C	C, Perl	C	C	C/C++

Figure 1.2.: Comparison matrix of the discussed benchmark/replay software

[1] Does not determine which files need to be pre created, but creates files that are explicitly added to a file "pool".

[2] Filepaths can be filtered or redirected to different physical files.

Key to the Feature Chart

A short description for each of the features in comparison chart.

Input The supported input (trace) file formats.

Preprocessing Defines if the replay tool uses some kind of preprocessing to reduce distortions during playback.

Binary Format Whether or not a binary format is supported.

Sanity check Is the trace checked for replayability in advance?

Precreation Defines if any files or other system state properties are reconstructed from the trace.

Expandable What mechanisms are provided to expand the application?

Modifiers Can the trace be modified and how. Offline modifications are applied before playback. Online modifications are applied during playback.

POSIX Whether or not POSIX system calls, and especially POSIX I/O calls are supported.

Parallel Is the software suitable for parallel applications applications? Either multithreading or multiprocessing with inter-process communication (IPC) support.

Threads Is there suppport for multithreading either built-in or by extensions?

MPI Is there support for MPI I/O and MPI IPC either built-in or by extension?

Open Source Defines if the source code is available for the public for inspection.

Dependencies What software dependencies are required on the host system?

Programming Language The programming languages used for the tool.

Conclusion of the Comparison

It becomes apparent that a first fundamental choice for replay software might be whether it is used for benchmarking of for forensics.

1.4. Goals of the Thesis, Requirements and Contribution

The goal of the thesis is to design and also implement a corresponding prototype for a comprehensive and *flexible event imitation engine* (feign). In contrast to existing replay tools, feign does not stop with trying to make scientific workloads more portable, but also aims to provide a virtual laboratory to exchange and experiment with workloads. The resulting key requirements include:

- Portability by eliminating dependencies.
- Modularity to support arbitrary (I/O) libraries.
- Support for parallel workloads
- And trace manipulation to adjust to other systems and for the virtual lab.

These requirements stem from the variety of different circumstances as outlined in the previous sections. The diversity of file systems and other technologies, that effect the performance of parallel systems make modularity a key requirement. This is further escalated by a huge variety of trace file formats. Support for different layers should be therefor be realized by providing a plugin architecture. Also strictly replaying a trace is often not adequate for the following reasons:

- The deterministic nature of traces, while usually a good property, is problematic for asynchronous interactions.
- Traces that have been recorded for another purpose can be polluted by a lot of unrelated activity.
- Some adjustments maybe be necessary to replay in a meaningful way on another system. This maybe adjusting the offsets of activity, or optimizations which need to be applied or retracted.
- Randomness/jitter or some other variation may be wanted for more realistic playback.
- Other replay approaches may be wanted, that could be derived from the trace, e.g. load-based replay.

All these use cases motivate the need for ways to manipulate the trace to improve meaningful playback. But manipulation of traces opens up new possibilities and can be used to create new knowledge automatically e.g. by using feign as a virtual lab to test different optimisations.

Summary and Structure of the Thesis

Chapter 1 introduced motivations for a comprehensive replay tool which is portable, flexible, expandable and suitable for scientific applications. It is also discussed why it should be possible to modify traces, and how this would allow a replay tool to be integrated as a virtual lab into autonomous optimisation engines. We will refine the requirements and propose a general design for a comprehensive replay tool in Chapter 2. Implementation details and how to create plugins, as well as the protoype implementations to support POSIX/MPI/SIOX are presented in Chapter 3. In Chapter 4 the evaluation of the prototype follows. And finally we summarize the results and discuss future work in Chapter 5.

2. Design

This chapter proposes a design which addresses the requirements derived from Chapter 1. Following a bottom up approach, we start with a compact design to support replay of arbitrary layers. The design is than expanded to allow environment pre-creation, automated trace manipulation and usage as a virtual lab.

2.1. Foundation for Flexible Replay

```
648848384, open, "myfile.txt", read-only, filehandle=4
649684854, read, 1024 bytes, ret=OK
649798378, read, 1024 bytes, ret=OK
...
654959594, close, filehandle=4
```

Figure 2.1.: A fictional example trace of a program reading a file.

Given a trace file, a replay tool needs to feature at least three logical components to replay a trace.

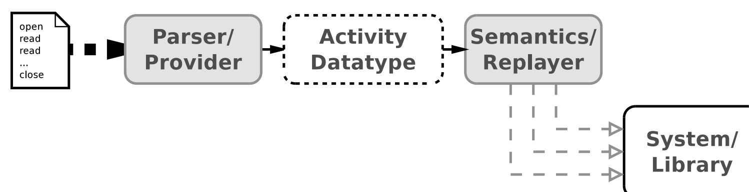


Figure 2.2.: Turning a trace into system or library activity.

- A *data type* to represent activities during runtime.
- A *provider* to read and parse entries from trace files and turn them into an activity data type.
- A *replayer* that knows the semantics of an activity and what is necessary to reproduce them.

If the providers and replayers can be exchanged, the source of the input and the resulting action solely depend on the availability of a matching plugin.

Provider plugins for common data sources could be developed, e.g. to support JSON, XML or CSV as well as database management systems such as postgreSQL or mongoDB. Furthermore, a provider could generate activity on the fly, either because it is hardcoded into the plugin or because the plugin supports an interpreter, e.g. for Lua [Lua, 2014] or Python.
The core responsibilities that remains for the replay tool are the coordination of replay and the provision of logistic support. To do so in an efficient manner, the replay engine needs to know some activity attributes such as the time or layer.

2.2. Activity Data Type

The requirements for an activity data type are conflictive. The diversity of different activities in trace files requires a dynamic data type favorable with support for nesting and reflection. Reflection would allow to enumerate over the members of the data type. But dynamic data types inflict memory and management overheads. For trace replay both, memory and processing time should be considered scarce.
Time can become a problem for rapid activity sequences. Anticipating trace manipulation, expensive mutations and filtering will add to the general overhead.
As trace files can be very large, we cannot assume that a complete trace can be buffered within the systems main memory (RAM). This is further complicated by the memory demands of plugins e.g. large writes in a replay plugin need to allocate memory or the activity cannot be adequately reproduced.
The consequences are disturbing because if the replayer issues I/O during playback to get new activities, distortions are added and the overhead maybe increased. There are some strategies to minimize these effects. Most likely, they add efficient serialization of our data type to the list of requirements.

2.3. Minimizing Distortions

As we cannot avoid distortions we need to try to minimize them as good was we can. One of the goals of feign is to make life for plugin developers as convienient as possible. So minimizing distortions that relate to the coordination of replay should be a responsibility of the replay engine. To minimize I/O, we need to ensure that activities are represented in a lightweight way and that only activity that is really replayed is fetched from the trace. Reducing the trace to the bare minimum can be achieved by "replaying" the trace twice. Or more precisely:

1. Create a stripped temporary trace from a full trace in a first run.
2. Replay the stripped trace.

If we have a serializeable activity datatype, we can turn it into a minimal binary representation. This temporal trace maybe only valid for a combination of `<architecture, trace, plugin configuration>`.

2.4. Replayability, Environment and State Management

When replaying a trace file we expect the replay to match the original. This includes that a call that succeeds in the trace also succeeds during replay. And that calls that fail, fail for the same reasons they did for the original. For I/O this may depend on the existence and size of files, for MPI we also have to consider MPI Datatypes and communication groups. It also becomes apparent that some operations depend on others to complete successfully. A simple example is that we cannot read or write from or to a file without having obtained a filehandle first.

2.4.1. Replayability

As replay is state dependant, it is useful to reason about a concept of replayability. The requirements to reach replayability may vary, but still some classifications can be made.

Direct Replayability

Activities come in an order such that all activities complete successfully. It can make sense to relax the successfull completion criterium and consider all traces directly replayable that e.g. do not crash the program.

Implicit Replayabilty

The tracefile might not be directly replayable but contains all the information to derive a directly replayable trace (e.g. by inserting a missing activity or by preparing the environment) This might require sophisticated programs or manual work.

Partial and/or approximate Replayability

Certain activities of the trace are not directly nor implicitly replayable. There might be two strategies to deal with such activity. Either drop them and derive a replayable trace, or by altering or inserting activities (however this time there are multiple solutions, that might differ from the actual application).

2.4.2. Environment Pre-Creation

Many traces are only implicitly replayable when replayed on another system. This is especially true for I/O traces. Files that have been available on the original system are unlikely to be present on another. It maybe acceptable to create a few files manually, but scientific applications may read from many sources. Since the sufficient file size for each opened file can be calculated automatically, pre-creation should be accounted for in the design of the replay engine.
As pre-creation during playback usually makes no sense due to distortions, the trace file needs to be processed in advance.

2.4.3. State Management during Playback

A replay plugin needs to perform some management tasks to replay a trace. A good example is the mapping of open file handles that occur in a trace to an actual file handle that is used during runtime. For file handles this is usually relatively easy (e.g. std::map), but none the less helpers for this could be part of a library available to plugins.

2.5. Trace Manipulation

Among the most pressing requirements to enable evaluation of different systems is trace manipulation. The replay engine itself further reinforced this requirement. (cf. Sections 2.3 and 2.4.1). Section 1.4 provided examples what kind of operations we would like to perform on a trace:

- filter/remove activities from the trace
- modify/mutate activities
- insert activities

As a decision to do any of these operations may depend on patterns, the operations need to be context aware. Plugins need to be able to traverse and alter the list of activities. All of the above operations to manipulate a trace could be conveniently achieved through one context-aware handler type if the order of events is represented by using a doubly linked list. If many plugins are used, always notifying every plugin can cost precious time. Again, more information for the replay engine allows for better performance. Therefor it should be considered to allow plugins to announce their intents at least on a per layer basis.
Section 2.3 is introducing the possibility to further distinguish between online (during playback) and offline trace manipulation. Many deterministic modifications can and should be made offline. But if jitter or modifications that depend on runtime information are wanted offline replay is very limited.

2.6. Easing Plugin Development

As already considered for some of the other sections, feign is built to make a daunting task more convenient. Over time common replay related problems and solutions should crystallize and could be integrated into a collection of tool chains and software libraries for others to built upon.
The most time consuming task when creating replay software becomes implementing support for layers. Each function call that produces activity of interest needs to be understood, the relevant parameters need to be extracted. A wrapper function has to be created, that prepares all necessary parameters. Because the call might depend on a preceding activity, some bookkeeping might be involved (e.g. for file handles). Finally, the wrapper can call the original function and handle the return value. Corresponding data

types and a helper to parse the activity from a trace file format might also be necessary. This approach does not scale. For example, the MPI standard [Passing and Forum, 2012] defines more than 280 functions.
A promising alternative is the semi-automatic approach used in SIOX to instrument library headers can also be used to create replay and provider plugins as was shown by TraceReplay [Ahlers, 2012].

2.7. Virtual Lab

One of the more innovative uses of feign is to integrate it with other software as a virtual lab. This becomes possible because plugins can be stacked in different ways to craft "new" tools. Figure 2.3 shows a minimalistic approach how such a lab environment could look like.

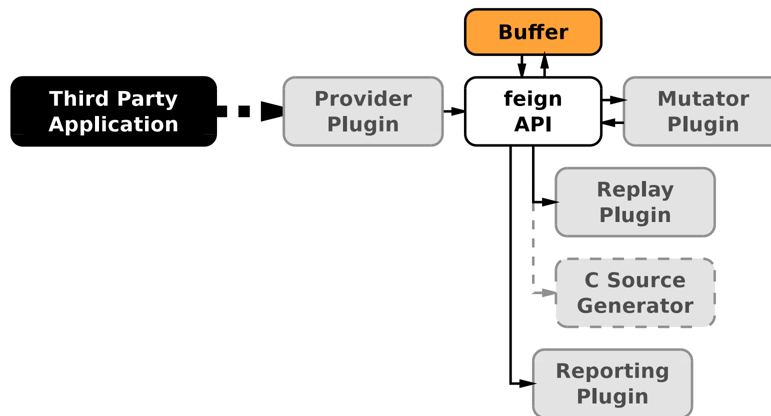


Figure 2.3.: Example stack to use feign as a virtual lab

Autonomous optimisations engines may have a collection of possible optimisation strategies. The optimisation engine may have found a pattern in an application but is unsure which optimisation results in the best performance. At this point the optimizer could turn to feign, set up an experiment, and simply test the different optimisations. A mutator plugin would apply the optimisation to the trace and communicate the result through an reporting plugin.
One interesting property for this approach is, that optimisations are usually applied based on activity monitored in the past. But in feign sophisticated strategies can be applied that also consider the events that come in the future.
To serve as a virtual lab, a well structured application programming interface API is needed. Tuning and configuration should be possible through arguments and environment variables.

Summary

Provider and Replay plugins, as well as a common data type have been identified as the central building blocks for a comprehensive replay engine for arbitrary layers in Sections 2.1 to 2.4. Distortions during playback can not be avoided for larger traces but pre-procession can minimize the problem (cf. 2.3). Sectionx 2.4 introduces the concept of replayability and explains why certain environment and runtime conditions need to be created to successfully replay a trace. Section 2.5 outlines how all trace manipulations could be done using one context-aware operation that is able to travers the activity buffer. But considering a more manifold approach might be more efficient. Section 2.6 advertises the development of a tool chain to ease or automate plugin creation. And Section 4.4 explains the virtual lab in more detail.

3. Implementation

This chapters covers the prototype implementation of the design developed in Chapter 2. The most important decisions are discussed and the application programming interface (API) is presented. It is then shown how to create a plugini ins Section 3.5. And finaly prototype plugins for POSIX, MPI and SIOX are covered briefly in Chapters 3.6, 3.7 and 3.8

3.1. Implementation Choices

Many implementation details are imposed by choosing a programming language. Because we target scientific applications that often use low-level system interfaces C/C++ are a natural choice. Large parts of the Linux kernel are implemented in C, and C has a standardized application binary interface (ABI). This is helpful for a plugin environment because programs do not need to be compiled the same way and still can communicate flawlessly. Also the latest MPI Standard Version 3 defines only C and Fortran bindings. Plugin systems in C and C++ are typically implemented by utilizing shared libraries that are dynamically loaded at runtime. While all the common platforms offer shared libraries in one way or another, there is no standardized cross platform approach. The prototype implementation will therefor work with dlopen() provided by most Linux distributions.
Higher level programming languages support can still be integrated later [Boo, 2014, Lua, 2014]. This is especially nice for plugins that filter, mutate as implementing such in higher level languages is often easier.

3.2. Activity Data type

The data type questions has not ultimately been decided. As C/C++ have no native data type that fulfills all the requirement from Section 2.2, Different approaches are possible but every one comes with certain drawbacks.
Tests with nested structs and a way to attach arbitrary data using void pointers have been made. The advantages of this approach is that it can be implemented in pure C, but serialization is complicated and would require extra effort from plugin creators.
As C++ allows polymorphism, using a Activity class seems to be a good idea. Dynamic casting allows to share activities but also to attach own data. This requires all plugins and feign to be compiled with the same compiler on the same system.
JSON [JSO, 2014], or BSON [BSO, 2014] are tempting because they are more schemaless data type. But they are not as memory efficient.

3.3. Feign: Overview of all Components

Figure 3.1 provides an overview of the different structural components of feign. To context-aware mutator plugins an iterator for the buffer is passed. Plugins can request to increase the lookahead.

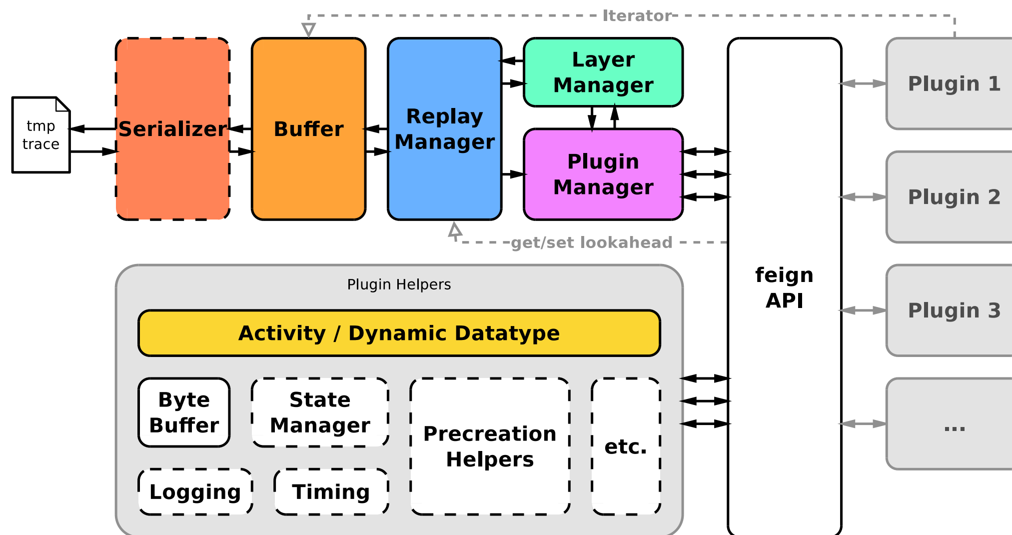


Figure 3.1.: The relation of different structural components of feign.

feign API Feigns application programming interface (API) defines how plugins can interact with feign. It also exposes the activity data type and the plugin helper library.

Plugin Manager The Plugin Manager keeps track of all loaded plugins, it is the central communication hub between components of feign and plugins. It also loads the shared libraries using dlopen and resolves the plugin handlers using dlsym.

Layer Manager The Layer Manager keeps track of the registered layers and maintains handler lists for the different hooks during the replay workflow. This way only plugins get notified who previously subscribed.

Activity / Dynamic Data Type The activity data type forms the basis for all replay related actions. Can attach their own data

Replay Manager The Replay Manager coordinates replay and

Buffer To realize lookahead and efficient replay.

Serializer The Serializer will write the temporary trace files from the pre-creation run and turns into an activity provider during playback.

3.4. Feign API and Internals

The feign API offers plugins to directly interact with feign. The plugin plugin API is C, with one exception (context aware plugins).

3.4.1. Plugin registration

When feign is called with `--plugin` and a shared object is provided feign will load it as a dynamic library using dlopen and resolve the init symbol using dlsym. Feign then calls init of the a plugin and the plugin can register itself and arbitrary layers.

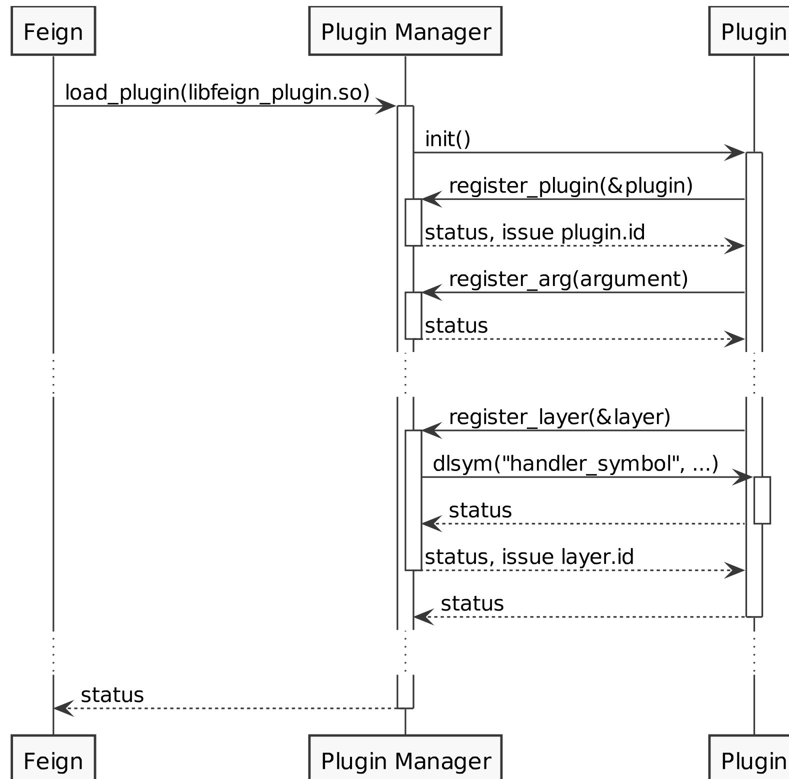


Figure 3.2.: Plugin registration

3.4.2. Layer registration

The following sequence diagram illustrates how to register a layer. The layer manager checks if a layer with the same name has already been registered. If so the runtime id of the layer is returned to the plugin, otherwise a new layer is created and then the runtime layer id is returned. Plugins can announce what they intent to do with the layer, e.g. filter.

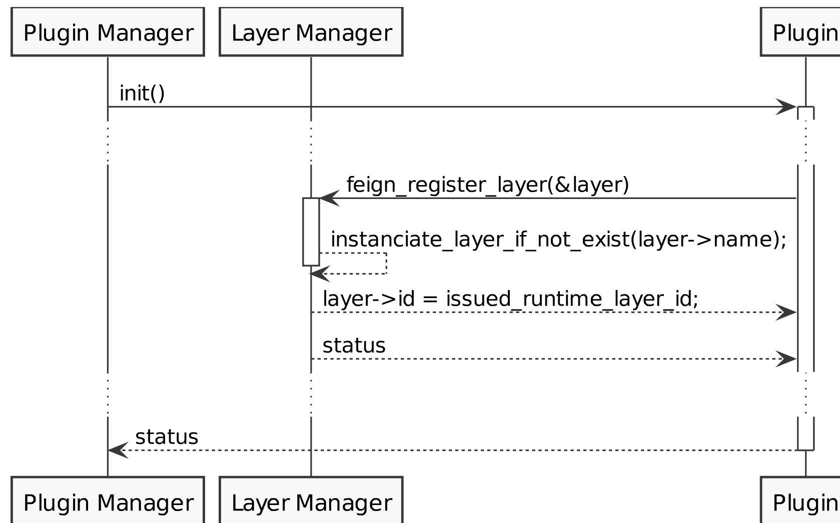


Figure 3.3.: Layer registration

3.4.3. The activity work flow

Feign establishes an activity work flow and allows plugins to hook into it various locations.

- providers/destroyers (as the provider is also responsible for destroy)
- context unaware filtes/mutators
- context aware filtes/mutators
- replayers / writers

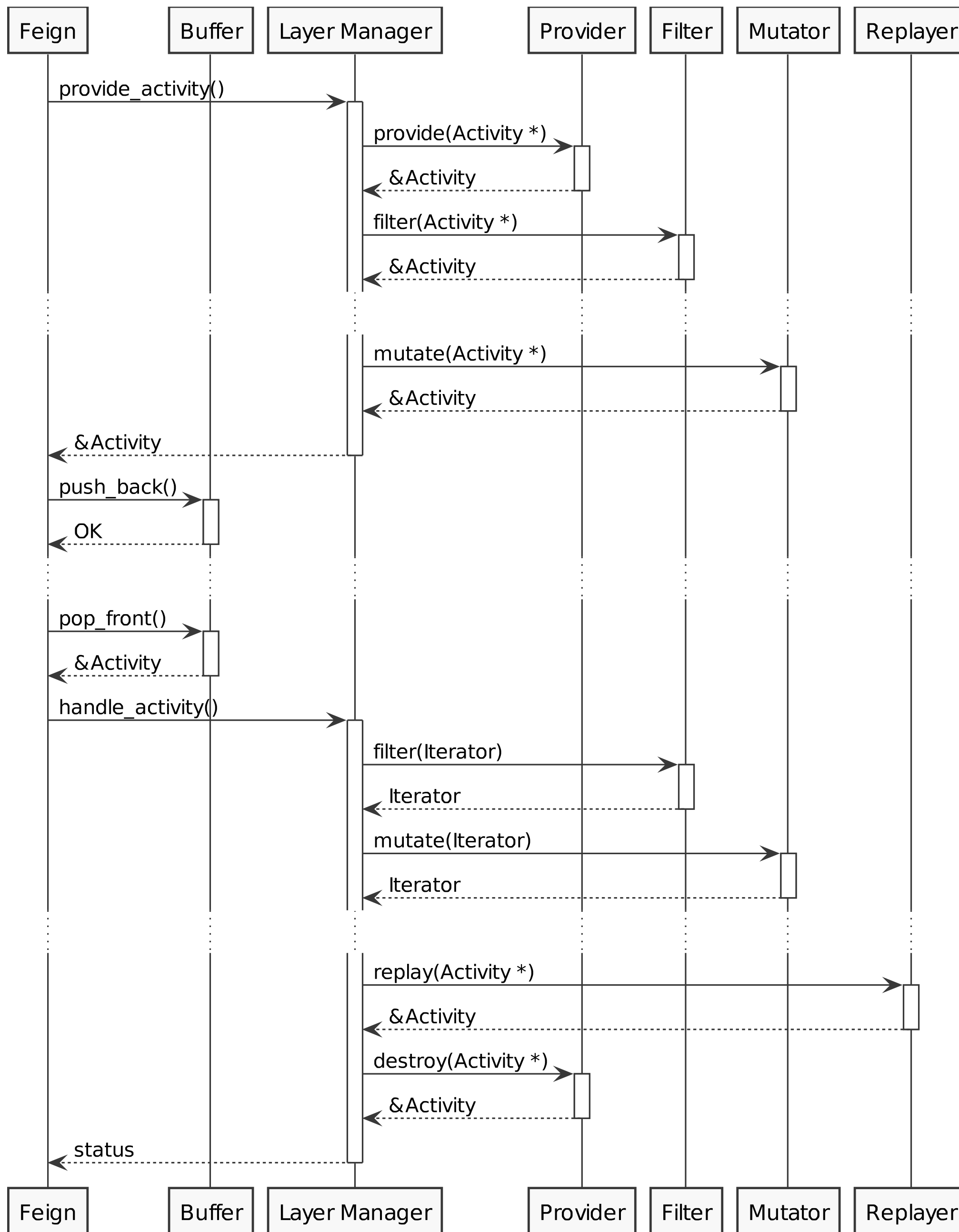


Figure 3.4.: Sequence diagram of the activity work flow

3.4.4. Timing

Feign wraps nanosleep which is used to create the offsets during playback. The wrapper is also used to account for feigns own overhead, and to dampen inaccuracies by the operating system. For example very short sleep periods are spent busy waiting. Sleep

may wake up early, if so, the wrapper issues to sleep again.

3.4.5. Plugin Helpers

As certain problems repeat during plugin development, facilities to offer plugin helpers are planned but out of scope of the thesis. But some helper functions e.g. to populate the filesystem as well as a shared buffer (e.g. for reads and writes) and a logger are provided.

3.5. Developing Plugins

Listing 3.1 demonstrates how plugins are implemented. The Example registers a plugin that can provide and replay activity for the layer "ExampleLayer".

```
#include <feign.h>

// information about the plugin
Plugin plugin = {
    .name = "Example-Provider-Replayer",
    .version = NULL,
};

// information about a layer
Layer layer = {
    .name = "ExampleLayer",
    .intents = FEIGN_PROVIDER | FEIGN_REPLAYER,
}

int init(int argc, char *argv[])
    // registration of plugin and layer
    feign_register_plugin(&plugin);
    feign_register_layer(&layer);
    return o;
}

// expected because of FEIGN_PROVIDER
Activity * provide(Activity * activity) {
    return create_activity_from_datasource();
}

Activity * destroy(Activity * activity) {
    if ( activity->provider == plugin.id ) {
        // this plugin created the activity, so it needs to destroy it
        free(Activity);
        return NULL;
    } else {
        return activity;
    }
}

// expected because of FEIGN_REPLAYER
```

```
Activity * replay(Activity * activity) {
    if ( activtiy->layer == layer.id ) {
        do_something();
        return NULL;
    } else {
        return activity;
    }
}
```

Listing 3.1: A example plugin

3.6. Supporting POSIX I/O

A datatypes header file is shared between the POSIX-plugins.

Provider The SIOX-Provider-Plugin is implemented as a provider for POSIX traces.

Precreator The POSIX-Precreator creates a list of all files that are opened, and calculates the minimal file size by keeping track of the read and write operations.

Replayer Subsets of the POSIX I/O (open, creat, read, write, close) and the buffered variants in stdio.h (fopen, fread, fwrite, fclose) have been implemented. A std::map is used to keep track of the different file handles. The plugin uses the ByteBuffer provided by feigns API.

3.7. Supporting MPI

As with the POSIX-plugins a data types header file is shared between the MPI-plugins. MPI

Provider The SIOX-Provider-Plugin is implemented as a provider for MPI traces.

Replayer/Precreation The MPI replay plugin introduced the need for a additional way to load plugins needs to be loaded using the `-plugin_global` option of feign. To prepare the environment for MPI I/O many parts of the logic can be reused from the POSIX pre creation plugin. The file size calculation will need to be adjusted to account for collective operations.

3.8. Supporting SIOX

The SIOX provider plugin uses the data type header files of the MPI plugins and the POSIX plugins.

Provider The SIOX-Provider-Plugin reads the binary trace format of SIOX, and turns SIOX activties into feign-activities that can get replayed with the POSIX- and/or MPI replayers. The provider will gracefully skip activity it does not know. As SIOX instruments using annotations in library headers that provide the necessary hints to generate suitable wrappers, a reverse approach should also work to generate this provider as well as replaying plugins for the different layers.

Summary

The most important implementation choices and their consequences have been discussed in Section 3.1. A combination of C/C++ was found to suit the the requirements formulated in Section 1.4 and Chapter 2 the best due to universal availability and proximity to low-level APIs. Sections 3.3 to 3.4 introduces the structural components of feign and provides some insight to internals. Section 3.5 covers plugin development, and finally a brief overview of the prototype layer is given.

4. Evaluation

This chapter evaluates how feign performs in respect to the goals specified in Chapter 1 and Chapter 2. The overhead is determined and accuracy of the replay is discussed in Sections 4.1 and 4.2.

4.1. Overhead

The overhead is the most critical performance metric for a replay eninge. It dictates what kind of workloads can be replayed in a meaningful way (cf. Section 4.2).

4.1.1. Test setup

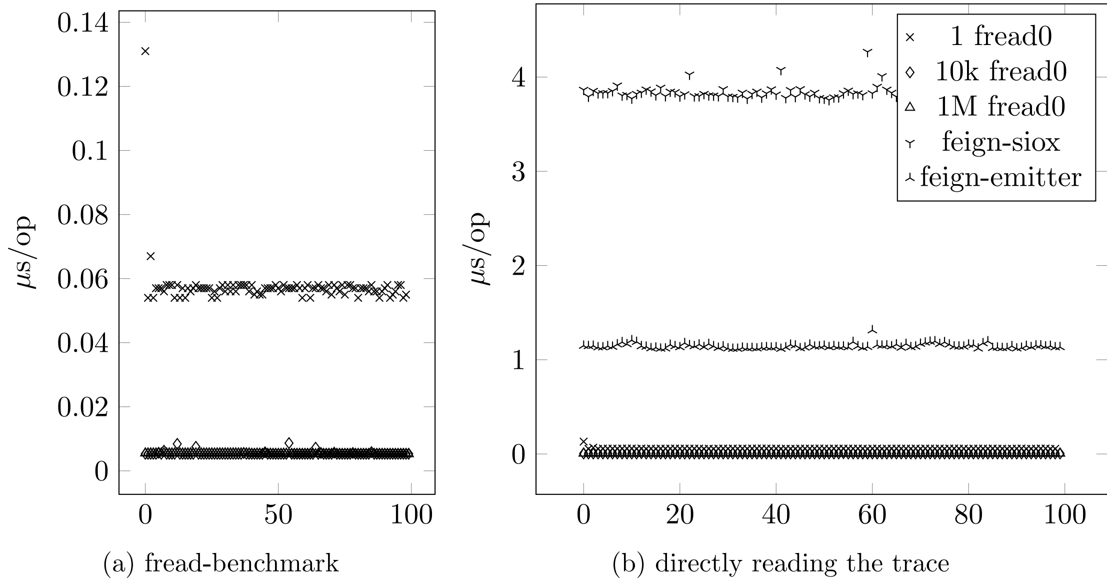
To determine the overhead feign introduces to replay one activity, the fread system call with a read length of zero was used, as it should return immediately and will remain unoptimized. The reference application and feign where compiled using the `-O2` compiler flag. `CLOCK_MONTONIC` has been used as a time source. For three variations of the fread0 scenario, and two feign setups 100 samples each have been collected on two different systems.

x **fread0** The micro benchmark that simulates rapid activity. `fopen, start timer, fread0 called` x `times, end timer, fclose`

feign-siox A SIOX trace for 10k fread0 was generated and the SIOX-provider is used to read activities from the trace. The results of this test are the bases for the overhead when replaying trace files.

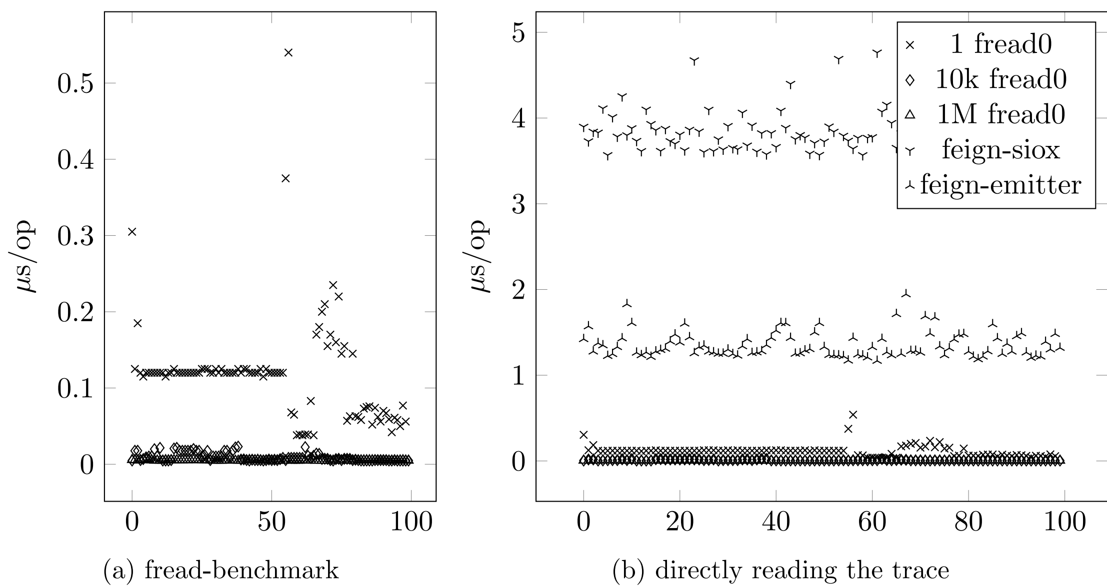
feign-emitter A provider plugin that provides 10.000 fread0-operations generated at runtime (No file I/O). As feign also allows plugins to act as workload generators, the results of this test deliver the bases for overheads to be expected by workload generators.

4.1.2. Results



(a) fread-benchmark (b) directly reading the trace

Figure 4.1.: Overhead comparison application and feign on the WR-Cluster



(a) fread-benchmark (b) directly reading the trace

Figure 4.2.: Overhead comparison application and feign on a consumer notebook

4.1.3. Test System

Measured was on two different systems, a consumer notebook and on the research cluster of the working group "Scientific Computing" located in the DKRZ (German Climate

Computing Center) in Hamburg, Germany.

- WR-Cluster with 10 nodes each featuring:
  - 2×Intel Xeon X5650@2.67GH
  - 12 GByte RAM
  - Seagate Barracuda 7200.12
- Apple Macbook A1370 (Mid-2011) (Ubuntu 13.10)
  - 1.8 GHz Core i7 (I7-2677M) (Boosts to 2.8GHz)
  - 1333 MHz DDR3 SDRAM

4.2. Accuracy of the Replayed Trace

Since all recorded activity is replayed with the right replay plugins, the original operations are reproduced. However, the temporal pattern of operations is also important. The overhead measurements impose limits to replay accuracy. Activity sequences with less than 4μs in between will be stretched, other activity will be fine as the overhead can be automatically be accounted for in feigns own sleep wrapper.

4.3. MPIOM

To show feign can replay the workloads of scientific applications a trace of the Max Planck Institute ocean model (MPIOM) [MPI, 2014] was generated using SIOX. While not all calls used in the application have been implemented in the POSIX- and MPI-prototype plugins, feign was still able to:

- load the trace data using the SIOX-provider and turn them into feign-activities for POSIX and MPI plugin prototypes
- negotiate which process replays which activity from the trace with the MPI-replayer/precreator
- precreate files using the POSIX-precreator
- replay the trace with the correct replay plugin, e.g. a POSIX activity with the loaded POSIX-replayer

Not implemented operations of MPI or POSIX were skipped. The trace could be replayed on two different systems (cf. 4.1.3

4.4. Using feign as a Virtual Lab

To show how feign could be used as a virtual lab a dummy application performing read operations on a 512MiB file is traced using SIOX. Feign is then run with 3 different plugins that injects posix_fadvise access hints if 5 successive reads are observed.

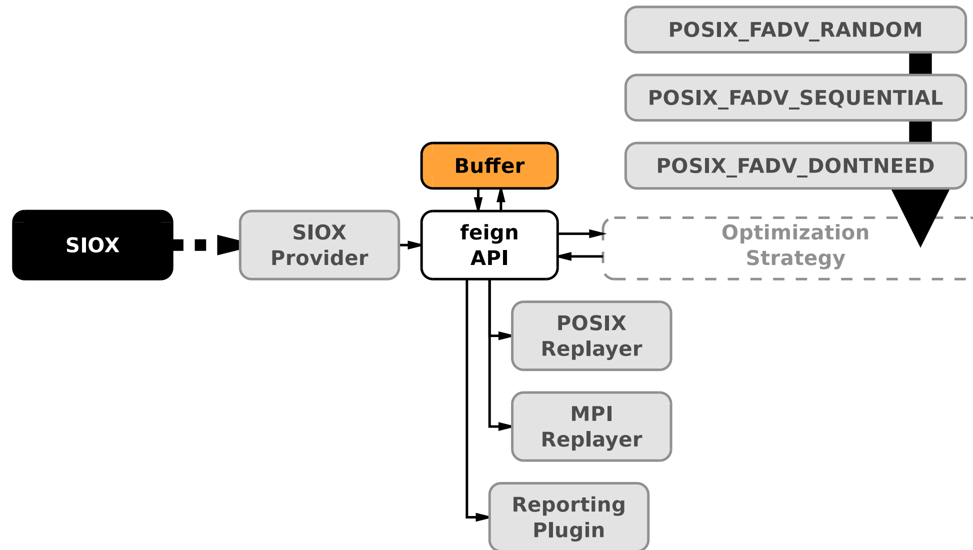


Figure 4.3.: Lab setup to try different optimisation strategies on a trace

The following output demonstrates how multiple experiments are made using different configurations to start feign.

```
$ export FEIGN_SIOX_TRACEDIR = ./trace

# unoptimized
$ feign-siox-posix.sh
us,us/op,ops/us
2477224.547000,247.722455,0.004037
2526341.169000,252.634117,0.003958
2518268.108000,251.826811,0.003971
...

# POSIX_FADV_SEQUENTIAL
$ feign-siox-posix.sh --plugin libfeign-inject_fadvise2.so
us,us/op,ops/us
2517719.168000,251.771917,0.003972
4605216.420000,460.521642,0.002171
2534644.459000,253.464446,0.003945
...
```

```
# POSIX_FADV_RANDOM
$ feign-siox-posix.sh --plugin libfeign-inject_fadvise1.so
us,us/op,ops/us
4717294.760000,471.729476,0.002120
2515886.380000,251.588638,0.003975
2519459.279000,251.945928,0.003969
...

# POSIX_FADV_DONTNEED
$ feign-siox-posix.sh --plugin libfeign-inject_fadvise4.so
us,us/op,ops/us
5194626.859000,519.462686,0.001925
5246762.685000,524.676268,0.001906
5103049.394000,510.304939,0.001960
...
```

The results is that none of the tried optimisations decreased the runtime. But at least the optimiser may refrain from applying POSIX_FADV_DONTNEED to a sequential read.

4.4.1. Dummy application

Listing 4.1

```c
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int i;
    int fd;
    ssize_t nr;

    char buf[1024*1024];

    fd = open("tmpfile", O_RDONLY );
    if ( fd != -1 ) {
        for (i = 0; i < 512; ++i)
        {
            nr = read(fd, &buf, sizeof(buf));
            if (nr == -1)
                perror("read");
        }

        close(fd);
    }

    return 0;
}
```

Listing 4.1: Dummy application reading from a file (512 MiB)

4.5. Comparing feign to the Related Work

Figure 4.4 revisits the comparison chart from Subsection 1.3.8 and appends feign. The prototype for feign proves that most of the features can be implemented using plugins.

	TraceReplay	Parabench	DIOS	feign
Input	Plugins	PBL Script	XML, Vampir	Plugins
Preprocessing	-	-	-	Plugins
Binary Format	-	-	-	Plugins[2]
Sanity Check	-	-	Yes	Yes
Precreation	-	-	Pool[1]	Yes
Extandable	Plugins	Interpreter	-	Plugins
Modifiers	Filters	-	Expressions	Plugins
offline	-	-	-	Yes
online	Yes	-	Yes	Yes
POSIX	Partial	Yes	Yes	Partial
Parallel	-	MPI	MPI	Plugins
Threads	-	-	-	-
MPI	-	Yes	Yes	Partial
IO	-	Yes	Yes	Partial
Comm	-	Barrier,Groups	Barrier	Partial
Userpace	Yes	Yes	Yes	Yes
Kernel-module	-	-	-	-
Open Source	-	Yes	-	Yes
Dependencies	Glib	Glib, Bison	Bison, libxml	none, Glib[3]
Prog. Language	C	C	C/C++	C/C++

Figure 4.4.: Revisiting the comparison matrix from Subsection 1.3.8

Summary

The overhead of feign has been determined in Section 4.1 and the resulting consequences for replay accuracy have been covered in Section 4.2. The result is that applications that do not undercut 4µs between operations can be replayed in time. To test if basic parallel trace replay and basic precreation of the files is working an scientifc application was trace and replayed on two differend systems.

[1] Does not determine which files need to be pre created, but creates files that are explicitly added to a file "pool".

[2] The SIOX-provider is using a binary format, a plugin to serialize feigns activities can be developed, but a built in solution for serialization would be favored.

[3] Glib was only used for argument parsing and can easily be removed.

5. Summary and Future Work

The preceding chapters handled the motivation, proposed a design and disccused a prototype implementation for a comprehensive replay tool. In this chapter the conclusions are summarized and future work is presented.

5.1. Summary

The thesis was able to demonstrate that a flexible replay engine that uses plugins to support arbitrary layers is capable of replaying parallel workloads that use MPI and POSIX I/O. Different ways to provide trace data have been used explored. Most notable, the successful implementation of the SIOX-provider to process SIOX traces suggest that that any data source can be supported for replay. Not the replayer but the tracing strategy is the determining factor for replay quality.
Still distortions during playback can not be avoided for larger traces but pre-processing a temporary trace can minimize the problem. (cf. 2.3). Section 2.4 introduced the concept of replayability and explains why certain environment and runtime conditions need to be created to successfully replay a trace. Pre-creation has been incorporated into the replay workflow, and basic strategies for POSIX I/O and MPI have been implemented. Thriving for meaningful replay, traces can be adjusted to different systems using filters or mutators to remove, alter or inject activities. The replay engine offers multiple way to do so. (cf. Sections 2.5 and 3.4.3). Context-free and context-aware variants are offered and can either applied during pre-procession or online during playback.
The most important internals of the prototype have been documented using component and sequence diagrams. Also an example plugin is discussed to show that creating plugins for feign is a simple task. (cf. Section 3.5).
While there is room for improvement the overhead introduced by feign is neglectable for all workloads that do not undercut 4μs between operations. And last but not least, the replay engine engine could be diverted to be used as a virtual lab to conduct some experiments on a trace.

5.2. Future Work

While the concept is demonstrated in this thesis, a lot needs to be done. A quick overview is given in the following list.

- Use the semi-automatic approach to generate layer plugins. Manual layer implementation is far to work intensive.

- Identify what helpers are needed for Providers, Mutators, Precreators and Replayers. This will also support the semi-automatic layer generation.
- Find a more suitable container for activity data to allow binary temporary traces for optimal playback.
- Investigate prefetch strategies to further reduce overhead.
- Support for multithreaded applications. However unlike MPI, support for multithreading might need to be built-in to the replay engine.

A. Source Code

The source code is published at:
`https://github.com/jakobluettgau/feign`

List of Figures

1.1. A trace file typically holds information about the ”order of” and ”details about” the activities recorded. 6
1.2. Comparison matrix of the discussed benchmark/replay software 9

2.1. A fictional example trace of a program reading a file. 13
2.2. Turning a trace into system or library activity. 13
2.3. Example stack to use feign as a virtual lab 17

3.1. The relation of different structural components of feign. 20
3.2. Plugin registration 21
3.3. Layer registration 22
3.4. Sequence diagram of the activity work flow 23

4.1. Overhead comparison application and feign on the WR-Cluster 28
4.2. Overhead comparison application and feign on a consumer notebook . . . 28
4.3. Lab setup to try different optimisation strategies on a trace 30
4.4. Revisiting the comparison matrix from Subsection 1.3.8 32

Bibliography

[BSO, 2014] (2014). Binary JSON. http://bsonspec.org. [Online; accessed 2014-03-14].

[Boo, 2014] (2014). Boost.Python. http://www.boost.org/doc/libs/1_36_0/libs/python/doc/index.html. [Online; accessed 2014-03-14].

[ior, 2014] (2014). ioreplay. http://code.google.com/p/ioapps/wiki/ioreplay. [Online; accessed 2014-03-01].

[JSO, 2014] (2014). JSON. http://json.org/. [Online; accessed 2014-03-14].

[LAN, 2014] (2014). LANL-Trace. http://institute.lanl.gov/data/software/. [Online; accessed 2014-03-04].

[Lua, 2014] (2014). Lua. http://www.lua.org/. [Online; accessed 2014-02-14].

[MPI, 2014] (2014). MPIOM. http://www.mpimet.mpg.de/en/science/models/mpiom.html. [Online; accessed 2014-03-06].

[PLF, 2014] (2014). PLFS Maps. http://institutes.lanl.gov/plfs/maps/. [Online; accessed 2014-03-06].

[TOP, 2014] (2014). TOP500. http://www.top500.org/. [Online; accessed 2014-03-06].

[Vam, 2014] (2014). Vampir Trace. http://www.vampir.eu/. [Online; accessed 2014-03-14].

[Ahlers, 2012] Ahlers, J. (2012). *Replay Engine for Application Specific Wokrloads*. PhD thesis, University of Hamburg.

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., Yelick, K. A., and Sciences, C. (2006). The Landscape of Parallel Computing Research: A View from Berkeley.

[Joukov et al., 2005] Joukov, N., Wong, T., and Zadok, E. (2005). Accurate and Efficient Replaying of File System Traces. In *Fourth USENIX Conference: FAST*.

[Kluge, 2011] Kluge, M. (2011). Comparison and End-to-End Performance Analysis of Parallel File Systems. (September).

[Kunkel et al., 2014] Kunkel, J., Zimmer, M., Aguilera, A., Mickler, H., Wang, X., Chut, A., Michel, R., and Weging, J. (2014). The SIOX architecture – coupling automatic monitoring and optimization of parallel I/O.

[Love, 2013] Love, R. (2013). *Linux System Programming (2nd Edition)*. O'Reilly.

[May and Livermore, 2005] May, J. and Livermore, L. (2005). Pianola: A script-based I/O benchmark. *International Symposium on Automated Analysis-Driven Debugging*, pages 69–76.

[Mesnier et al.,] Mesnier, M. P., Wachs, M., Sambasivan, R. R., Lopez, J., Hendricks, J., and Ganger, G. R. //TRACE: Parallel trace replay with approximate causal events. pages 153–167.

[Mordvinova et al., 2010] Mordvinova, O., Runz, D., Kunkel, J., and Ludwig, T. (2010). I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. *Procedia Computer Science*, pages 2119–2128.

[Passing and Forum, 2012] Passing, M. and Forum, I. (2012). MPI : A Message-Passing Interface Standard.

[Sivathanu et al.,] Sivathanu, S., Kim, J., Kulkarni, D., Liu, L., and Alto, P. Load-Aware Replay of I/O Traces. Technical report.

[Thompson, 1980] Thompson, C. D. (1980). *A Complexity Theory for VLSI*. PhD thesis, Pittsburgh, PA, USA. AAI8100621.

[Zhu et al.,] Zhu, N., Chen, J., Chiueh, T.-c., and Brook, S. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation.

[Zimmer et al., 2012] Zimmer, M., Kunkel, J. M., and Ludwig, T. (2012). Towards Self-Optimization in HPC I / O.

Ich versichere, dass ich die Bachelorarbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

____

Ort, Datum, Unterschrift