Informatics

Session Recording in Configuration Management Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Maximilian Irlinger, BSc
Matrikelnummer 01426708

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, BSc

Wien, 20. Juni 2023

Maximilian Irlinger Jürgen Cito

Informatics

Session Recording in Configuration Management Environments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Maximilian Irlinger, BSc
Registration Number 01426708

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, BSc

Vienna, 20th June, 2023

Maximilian Irlinger   Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Maximilian Irlinger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Juni 2023

Maximilian Irlinger

Acknowledgements

I would like to express my deepest gratitude to my advisors, Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito and Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, for their exceptional guidance and unwavering support throughout this thesis. Their expertise, insightful feedback, and constant motivation have been instrumental in shaping my academic journey and pushing me towards excellence.

I am truly thankful to my parents and family for their unwavering love, understanding, and support. Their belief in me and their constant encouragement have been a great source of strength and inspiration during this challenging process.

I would also like to extend my sincere appreciation to the members of the Elektra Initiative. Their contributions, collaboration, and shared passion have been invaluable. Working with such talented individuals has been a rewarding experience, and I am grateful for the knowledge and friendship we have built.

I am deeply grateful to everyone who has played a part in my academic journey, including my mentors, family, friends, and project group members. Your support, guidance, and contributions have been essential, and I am honoured to have had the opportunity to work with such exceptional individuals.

Kurzfassung

Diese Arbeit präsentiert Record Elektra, ein Session-Recording-Tool, das auf Konfigurationsframework Elektra aufbaut und entwickelt wurde, um den Prozess der Anwendung von Konfigurationsänderungen auf mehreren Hosts zu optimieren.

Durch eine Fallstudie demonstrieren wir den Konfigurationsprozess unter Verwendung von Record Elektra und zeigen dabei die Wirksamkeit des Tools bei der Vereinfachung von Konfigurationsänderungen und der Reduzierung von Fehlern. Darüber hinaus haben wir die Leistungs- und Speicherauswirkungen von Record Elektra untersucht und festgestellt, dass sie akzeptablen Grenzen entsprechen und somit die Nutzung von Elektra nur minimal beeinträchtigen.

Unsere Forschung trägt zum Bereich der Konfigurationsverwaltung bei und betont den Wert des Session-Recordings zur Verbesserung des Konfigurationsprozesses für Software-Services. Die Studie konzentriert sich auf Änderungen, die direkt über Elektra vorgenommen werden, aber unser Ansatz bietet eine Grundlage für die Erweiterung des Session-Recordings, um Änderungen an Konfigurationsdateien direkt zu überwachen.

Abstract

This thesis presents Record Elektra, a session recording tool built on the configuration framework Elektra, designed to streamline the process of applying configuration changes across multiple hosts.

Through a case study, we show the configuration process of an embedded system using Record Elektra, demonstrating the tool's effectiveness in simplifying configuration changes and reducing errors. We also evaluated the performance and memory impact of Record Elektra, finding that it remains within acceptable limits, ensuring minimal disruption to the usage of Elektra.

Our research contributes to the field of configuration management, highlighting the value of session recording in enhancing the configuration process for software services. The study focuses on changes made directly via Elektra, but our approach provides a foundation for extending session recording to monitor changes made to configuration files directly.

Contents

CHAPTER 1

Introduction

With an ever-growing demand for services and compute resources, and reliability thereof, system administrators and operators find themselves deploying more and more software systems. One big challenge faced both during initial deployment and ongoing maintenance is keeping the configuration of the services consistent across multiple instances.

Configuration Management and DevOps tools such as Ansible [1], Chef [2] and Puppet [3] are often used to configure and manage a range of systems running on bare-metal or in virtual machines. Over the past decade, containerization using tools such as Docker [4] and Kubernetes [5], has become one of the big deployment strategies for both cloud and on-premise services.

While both – configuration management and containerization tools – make it way easier to deploy and manage services redundantly over multiple instances, there is a key area those kinds of tools fail to make easier: creating the configuration for the various applications. Sure, both sets of tools allow rolling out the created configuration, but this is only half the rent. The operator still has to write the configuration files for each one of the deployed applications. What makes matters worse is that almost every application uses a different configuration file format.

The open-source library *Libelektra* [6] (further called *Elektra*) and the accompanying Elektra Initiative set out to solve the latter problem. Elektra aims to provide configuration data of the system in a global, hierarchical key database. It is very extensible and has a sophisticated plugin interface. One type of plugin are so-called *storage plugins*, which are used as adaptors to read and write configuration files of existing applications. This way, even applications that do not directly integrate Elektra can be managed using it.

During the maintenance phase of the service life-cycle, changes to the configuration of the deployed application are often necessary. Some of those changes may be specific to certain nodes, while other changes need to be rolled out to all nodes running the application. Performing such changes manually is error-prone and time-consuming.

To mitigate these problems, we introduce the Record Elektra tool. Using this tool, administrators can record configuration changes on a single host. After the new configuration is complete, the recorded changes can be exported to and imported on all target machines.

We hypothesise that using this tool will streamline the process of applying iterative generated configuration changes on other hosts.

1.1 Aim of this Thesis

As mentioned above, we aim to show that session recording in configuration management tools is a valuable tool. To show this, a session recording tool for Elektra was developed.

While session recording sure can be a great feature to have, we do not expect the majority of users of Elektra to use it. Ensuring that this feature does not have any severe negative effects on their usage was of utmost importance.

In this thesis, we will concentrate on answering the following key research questions:

Research Question 1 *In which way does Record Elektra benefit the use case of rolling out changes of configuration?*

Research Question 2 *What performance and memory impact does Record Elektra have compared to vanilla Elektra?*

1.2 Methodological Approach

To evaluate the research questions, we developed a session recording plugin named Record Elektra. Before this could be done, we had to create the necessary infrastructure to support global plugins within Elektra. To enable a user-friendly workflow, tooling for starting recording sessions and exporting the changes also was developed.

For evaluating Research Question 1, we conducted a case study where a real-world setup consisting of multiple services was configured. During the case study, the configuration and reconfiguration to an in-advance determined known configuration of these services was done once using Record Elektra and once by manually editing the configuration files.

We evaluated Research Question 2 by creating different configurations of varying sizes, and comparing the runtime and memory footprint of Elektra instances which have Record Elektra enabled and disabled.

1.3 Scope and Limitations

For the session recording part, we are focusing only on changes that are done directly via Elektra. More specifically, we only record changes performed by either the command-line utility or the API. This has the added benefit that Elektra is able to check whether the configuration conforms to certain specifications [7].

Although our present analysis is limited to these specific methods, we have established a solid foundation that can be extended to monitor changes made to configuration files directly.

1.4 Structure of this Thesis

In Chapter 2, we delve into the essential background of the concepts and technologies used for our approach, as well as explore related work in this area, laying a solid foundation for our research.

Chapter 3 is dedicated to presenting the essential requirements we have formulated for our approach, ensuring that our solution addresses the specific needs and objectives of our study.

Chapter 4 takes a deep dive into the implementation details of our approach, providing a comprehensive and detailed description of how we have executed our solution, highlighting the technical aspects and methodologies employed.

Chapter 5 showcases the conducted case study, where we apply our approach to a real-world scenario, demonstrating its effectiveness and practical application in a tangible context.

In chapter 6, we present an overview of the benchmarks conducted on Record Elektra, providing a comparative analysis and insightful results that evaluate the performance and capabilities of our approach.

Chapter 7 evaluates whether our requirements have been successfully met, conducting a thorough assessment that examines how well our approach aligns with the initial set of requirements and criteria established.

Finally, in chapter 8, we conclude this thesis by summarizing the key findings, reflecting on the implications of our research, and offering an outlook on potential future work, thereby contributing to the ongoing development and advancement of the field.

CHAPTER 2

Background

2.1 Configuration Management

As best described by Burgess and Couch [8], *configuration management is the process of constraining the behavior of a network of machines so that each machine's behavior conforms to predefined policies and guidelines and accomplishes predetermined business objectives*. This also conforms with the *system configuration problem* that has been described by Anderson [9].

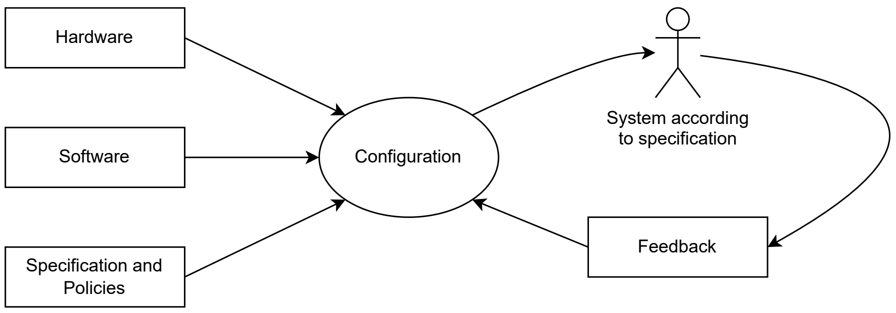


Figure 2.1: The System Configuration Problem as described by Anderson [9]

The software configuration problem, as illustrated in Figure 2.1, involves a multi-step process. Initially, the problem requires configuring several machines, installing the necessary software packages, and specifying the required services according to predetermined guidelines. Subsequently, feedback is collected, and adjustments are made to the machines' configurations based on the feedback or whenever any of the inputs, such as hardware, software, or specifications, undergo a change. This iterative process ensures that the machines are continuously updated to reflect the evolving needs of the system.

One of the first descriptions of dynamic configuration management with a central database was given by Anderson [10]. Other early theoretical groundwork was laid by Burgess [11], where he introduces the tool *CFEngine* [12]. Burgess's work was instrumental in laying the groundwork for modern configuration management approaches, which continue to play a critical role in managing complex systems today.

2.2 Elektra

The Elektra Initiative aims to solve the non-trivial issue of abstracting configuration for improved integration and reconfiguration of software. They make this possible with the library aptly named Elektra [13, 6].

Elektra describes itself as *the configuration framework for everyone.* At its core, Elektra is a global key database with various plugins to mount configuration files of numerous applications. It also provides a command-line tool called *kdb* that can be used to modify the configuration database.

In their study, Raab et al. [14] have shown, that developers using Elektra are significantly faster in common configuration management scenarios compared to other methods.

2.2.1 Keys and Namespaces

Elektra's global key database (KDB) is laid out hierarchically. Keys are the eponymous components of key databases. The following are example keys in Elektra :

```
system:/sw/apache/httpd/#0/current/num_processes
system:/sw/apache/httpd/#0/current/AllowOverride
user:/sw/kde/kicker/#0/current/preferred_applications/x-www
user:/sw/kde/kicker/#0/current/preferred_applications/x-mail
proc:/env/HOME
proc:/env/PWD
```

Keys in Elektra are hierarchical. Each hierarchical step is delimited by a slash (/). Figure 2.2 depicts the above keys as a hierarchical tree. A key may also have a value attached to it. Values can either be in text or binary form. Keys may also possess metadata, such as default values and data type constraints.

Every key exists in a namespace, e.g. `system:/`. Currently, 5 namespaces are available, each one serving slightly different purposes. The namespaces are listed in their default order:

- `spec:/` does not contain values but only metadata.
- `proc:/` read-only namespace that derives its data from the current process, such as program name, arguments and environment variables.

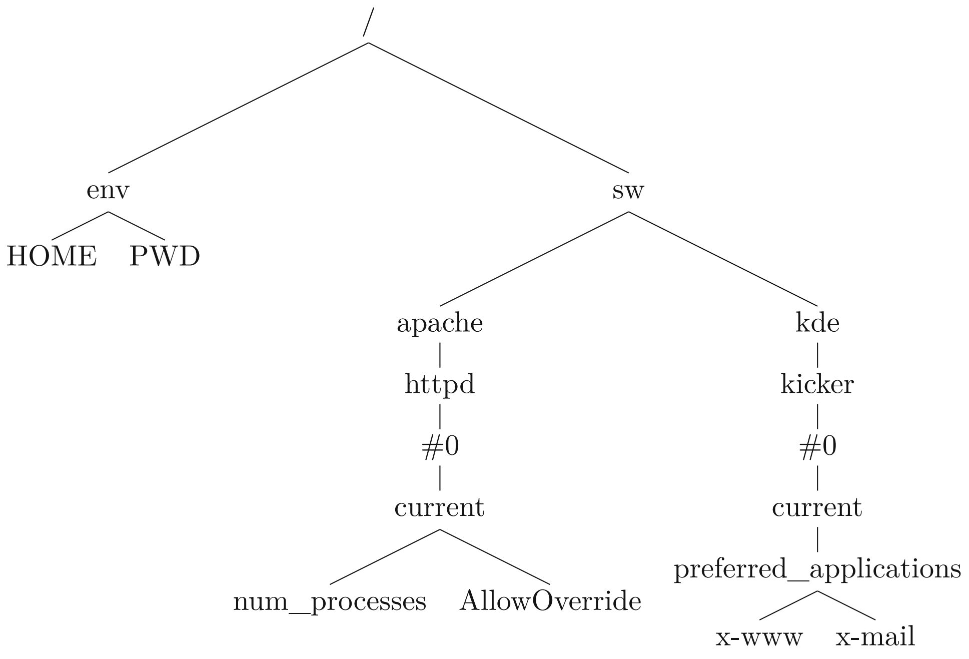


Figure 2.2: Tree representation of key database

- `dir:/` derives its values from the current working directory. Useful for e.g. working with `.htaccess` files in multi-tenant web hosting installations.

- `user:/` stores configuration on a per-user basis. This way, every user on a multi-user system can have their own custom configuration.

- `system:/` contains system-wide configuration.

When setting values, the namespace has to be explicitly provided. When reading values, the namespaces are optional. If no namespace is provided, Elektra will perform a *cascading* lookup.

During a cascading lookup, Elektra will cycle through the namespaces in the order described above. As soon as it finds the key, it returns the result. This mechanism allows users, e.g., to overwrite system configuration (stored in `system:/`) for their own profile (stored in `user:/`).

One notable exception are keys in the `spec:/` namespace. This namespace holds metadata that influences the cascading search. The search itself, however, usually is in the keys of the other namespaces. Only if it did not find a value for the key in another namespace, Elektra will return the default value that has been specified in the `spec:/` namespace.

2.2.2 The `kdb` Utility

The command-line `kdb` utility is one of the main ways for a user to interact with the key database. At the most basic level, `kdb` is used to set and retrieve values for keys, and their metadata.

- `kdb get <key>` retrieves the value of the given key.
- `kdb set <key> <value>` sets the value of the key.
- `kdb rm <key>` removes the key.
- `kdb ls <key>` lists all keys in the hierarchy below the specified key.
- `kdb meta-get <key> <meta name>` retrieves the metadata named `<meta name>` for the specified key.
- `kdb meta-set <key> <meta name> <meta value>` sets the metadata named `<meta name>` for the specified key.
- `kdb meta-rm <key> <meta name>` removes the metadata named `<meta name>` for the specified key.
- `kdb meta-ls <key>` lists all available metadata for the specified key.

The above list of commands is far from complete. A full list of available commands is available in the manual pages.

2.2.3 Mounting

The preferred way of making applications work with Elektra is to *elektrify* them. This means that the application's developers have to directly use the APIs that Elektra provides. In cases where this is not possible, Elektra provides a simpler way to integrate with the application. This approach is called *mounting* the configuration files of the application.

When mounting a configuration file, administrators tell Elektra that a specific subtree of the key database is stored in a particular file. A requirement for this is that a storage plugin for the format of the configuration file exists. Some of the default storage plugins shipped with Elektra include plugins for host files, Mozilla configuration files, XML-, JSON- and YAML files.

The following command mounts the hosts file into the subtree `system:/hosts` of the key database:

```
sudo kdb mount --with-recommends /etc/hosts system:/hosts hosts
```

The developer of the storage plugin has to decide how to map the contents of the configuration file to an Elektra key hierarchy. For the `hosts` plugin, the developers decided to split it up into IPv4- and IPv6-specific hostnames.

```
system:/hosts/ipv4/www.example.com -> 93.184.216.34
system:/hosts/ipv6/localhost       -> ::1
```

2.2.4 Extensibility

Storage plugins are not the only type of plugins for Elektra . In recent years, multiple research projects were conducted on how to integrate Elektra into configuration management tools. Denner created a plugin for interaction with Puppet [15] and Waser created an Ansible plugin [16].

Being a configuration-management tool, the plugins and their interactions themselves are configured using Elektra. One of the most useful plugin types are filter plugins. They receive configuration settings and can inspect their values. For example, checker plugins can be used to validate the format of IP addresses.

2.2.5 Validation and Specifications

Verifying configuration stored in Elektra has also been researched by Raab [7], allowing for immediate verification of whether the merged configuration is valid. This feature alone can be responsible for huge cost savings in a real-world deployment, as system downtime due to misconfiguration can be largely avoided [17].

The basic building block of validation in Elektra are the aforementioned filter plugins. Most of them can be configured using metadata on the keys themselves. The following scenario tells the `check` plugin that the value for the key `user:/tests/together/test` must be a number:

```
kdb meta-set user:/tests/together/test check/validation
 ↪  "[1-9][0-9]*"
kdb meta-set user:/tests/together/test check/validation/match
 ↪  LINE
kdb meta-set user:/tests/together/test check/validation/message
 ↪  "Not a number"
```

Being able to configure validation rules for every key is a very powerful feature. In practice, however, it would be cumbersome if the administrators of a system had to create those rules all by themselves. Also note that in this example, the validation metadata is stored *directly* on the keys. If the key would exist in multiple namespaces, e.g. `system:/` and `user:/`, those rules would have to be specified for every namespace.

This is where *specifications* come into play. Specifications live in the `spec:/` namespace. As mentioned in the section about namespaces, this namespace only contains metadata. metadata for a key in this namespace is automatically applied to the same key in all namespaces. This allows the creation of schemata by the application developers.

2.3 Ansible

Ansible [18, 1] is one of the most popular open-source configuration management tools available. One of its distinctive properties is that, compared to other tools such as CFEngine, Chef and Puppet, it is *agentless*. In short, this means that there is no special software required on the machines you want to manage. The only hard requirement is that the machine needs to run an SSH daemon.

Agentless tools have some key advantages over their counterparts requiring the installation of agent software on the target machines:

- Easy bootstrapping. There is no need to install and configure the agent for your configuration management tool.
- Less maintenance. You don't have to keep the agent up-to-date with the latest feature and security updates.
- Enhanced security. Every additional piece of software running on a machine is a potential attack vector. By not having something running constantly in the background, you automatically decrease the likelihood of your systems getting hacked.

Another key differentiator of Ansible is that it is *serverless*. If you're using Ansible, you don't need a central configuration server.

This has another few advantages:

- No extra infrastructure. You don't need to provision and set up extra hard- and software services just to manage your configuration.
- Improved scalability. As there is no central server, there are a lot fewer scalability issues when you're starting to configure hundreds of thousands of machines.
- Less maintenance. You do not have to maintain, update, backup, monitor, etc. the configuration servers.
- Enhanced security. As with agentlessness, having less software running minimizes the potential attack vectors your infrastructure presents to potential hackers.

2.3.1 Infrastructure As Code

Ansible embraces the concept of *infrastructure as code*. This is an approach, where system configuration and infrastructure automation is based on best practices from software development.

One key principle is that configuration is kept as human-readable text files, which can be easily versioned using common version control systems such as git. Therefore, the configuration and its history are always auditable.

2.3.2 Inventory

To configure your machines we need to tell Ansible which machines to manage. This is where the *inventory* comes into play. In its simplest form, the inventory is just a flat file called *hosts* which contains one host per line, as shown in Listing 2.1.

```
vienna.example.com
munich.example.com
paris.example.com
london.example.com
```

Listing 2.1: Basic inventory with four hosts

After the host name, Ansible allows the specification of variables for this host. Those variables can be some of the Ansible-internal variables, such as the SSH username, or completely custom variables. In the example depicted in Listing 2.2, we set the Ansible-internal variable *ansible_user* and the custom variable *web_server_port*.

```
vienna.example.com ansible_user=hans web_server_port=8080
munich.example.com ansible_user=franz web_server_port=8081
```

Listing 2.2: Inventory with variables for each host

Hosts can also be grouped. These groups can also overlap, and be nested. This allows for fine-grained control over which configuration is applied to which hosts. An example of this can be seen in the following Listing 2.3.

```
[web]
vienna.example.com
munich.example.com

[database]
paris.example.com
london.example.com

[accounting]
vienna.example.com
paris.example.com
```

Listing 2.3: Inventory with groups

The inventory doesn't have to be a flat file. It is also possible to use YAML and JSON files to describe the inventory. Ansible also has a neat way of dealing with *dynamic inventories*. If you already have a system in place where you've registered all your hosts, Ansible has a way query this information. You just need to provide it with an inventory file that is marked as *executable*. Ansible will execute this file and interpret the output of the execution as the inventory.

There are also numerous plugins to use other forms of inventory, such as Amazon EC2 and Azure Resource Manager.

2.3.3 Playbooks

The central component of configuration management with Ansible is the so-called *playbook*. A playbook is a configuration management script in YAML format. It describes the desired state of a system and the steps, represented by *tasks*, that should be taken to achieve that state.

Ansible playbooks are designed to be *idempotent*. Running the playbook multiple times should result in the same end state. This allows for safe and repeatable automation of IT tasks.

This idempotency, however, isn't something that Ansible can just force. It is also partly the users' responsibility to keep the playbooks idempotent. If instructed to, Ansible will happily change the same configuration in varying ways multiple times within the same play.

For example, a very simple playbook looks something like the following depicted in Listing 2.4. This example playbook consists of a single *play* named *Configure Webservers*. This play is executed on all hosts in the group *web*, as defined in the inventory in Listing 2.3. The first task in the play is named *Install nginx server*. It uses the *package module* to make sure that the package *nginx* is installed and up-to-date.

The second task *Copy nginx configuration* uses the *template* module to copy the file *nginx.config.j2* to its destination in */etc/nginx/nginx.conf*. If Ansible detects that this task changed the system configuration, it will notify the *handler* called *Restart nginx*.

A handler is a special type of task that is triggered only when a particular event occurs. Handlers are used to perform actions that are dependent on the outcome of other tasks, such as restarting a service when a configuration file has been modified. They are executed only once, at the end of the playbook run, after all other tasks have been completed. This ensures that all changes have been made before the handler is executed.

In our example, there is only a single handler. When triggered, it will use the *service* module to restart the nginx service.

```
---
- name: Configure Webservers
  hosts: web
  tasks:
    - name: Install nginx server
      package:
        name: nginx
        state: latest
    - name: Copy nginx configuration
      template:
        src: nginx.config.j2
        dest: /etc/nginx/nginx.conf
      notify: Restart nginx
  handlers:
    - name: Restart nginx
      service:
        name: nginx
        state: restarted
...
```

Listing 2.4: Example Playbook

2.3.4 Modules

An Ansible *module* is a standalone script or program that performs a specific task on a target host. Modules are the building blocks of Ansible. They are supposed to be idempotent, however, it is up to the module authors to ensure this.

While Ansible already includes hundreds of modules, it is possible to create custom ones to extend its functionality. This capability allows users to tailor Ansible to their specific needs and automate tasks that are not covered by the built-in modules. To simplify the process of sharing and discovering Ansible modules, there is a community-driven repository called *Ansible Galaxy*[1].

Creating a Custom Ansible Module

Modules can be written in any programming language that can be executed on the target host, such as Python, Ruby, Bash, or PowerShell. They can be executed locally on the target host or remotely through a connection established by Ansible. The most widely used programming language for Ansible module is Python.

An Ansible Python module is a standalone Python script that follows a specific structure. It typically consists of a module file written in Python and optionally, a module utilities file. These files reside in the module library directory on the control machine.

Modules accept input arguments that define their behaviour. These arguments can be passed to the module through the Ansible playbook or command-line interface. The argument specification within the Python module defines the parameters accepted by the module, including their names, types, descriptions, and default values. For Python modules, the arguments in the playbook are serialized as JSON and passed to the module on execution.

During execution, Ansible creates a compressed zip file that includes the module file, any imported utilities, and necessary boilerplate code. This zip file is then copied to a temporary directory on the managed host. The module file is extracted and executed, while the utilities and boilerplate code from the zip file are appended to the *PYTHONPATH* environment variable.

Ansible modules should return output data to the Ansible framework. This output data should be in JSON format and printed to *stdout*. The results can include any information relevant to the module's execution, such as status messages, configuration changes, or data retrieved from remote systems.

2.4 Related Work

In this section, we provide a short overview of related work in this field, which builds the foundation and inspiration for our work.

2.4.1 Git and EtcKeeper

A currently widespread procedure is having all configuration files on a system (i.e. the */etc* directory) tracked in a version control system like git[2]. This allows easy backup and restore and synchronisation across multiple systems. A commonly used tool suite for this approach is *etckeeper* [19], which integrates a bit deeper into the system and also does things like auto-commits on system updates.

[1]https://galaxy.ansible.com/
[2]https://git-scm.com

2.4.2 Semi-structured Merge

Managing multiple systems using the git approach can sometimes lead to merge conflicts. This is incredibly likely if changes to the same sections in configuration files are made on different systems.

Apel et al. [20] present the concept of a semistructured merge. In their approach, they don't only rely on textual differences for merges, but also include the syntactic and semantic meanings of the changes.

This is shown to reduce the number of merge conflicts by up to 60%. Cavalcanti et al. [21] also evaluated this approach for its impact on productivity and correctness of the merging process.

They also proposed and implemented further improvements to reduce the conflicts by 50%. Sousa et al. [22] introduce a merge verification approach based on the notion of semantic conflict-freedom, that is able to verify the correctness of the merge in 75% of the studied cases.

2.4.3 Docker Record

A session recording tool for Docker, aptly named *docker-record*, has been developed by Cito [23, 24]. This tool was born out of the needs and frustration developers suffer from when creating Docker containers.

Like system configuration, the definition for a Docker container is often created in iterations with much trial and error. Docker-record helps developers to record all interactive changes they performed within a container and then allows them to export a new Dockerfile with all those changes incorporated.

CHAPTER 3
Requirements

Before any implementation or case study can be done, we need to define the requirements we have for session recording and iteratively creating system configuration.

3.1 Functional Requirements

Functional requirements describe the results of the behaviour of the software system [25]. As such, here we describe the list of features we find are a must-have to enable an iterative workflow in configuration management.

FR 1 *Record any changes made to the configuration*

The main requirement is that the solution must be able to record all the changes made by the user. This includes changes made to any part of the configuration, regardless of the complexity or size of the configuration.

FR 2 *Record a subset of the configuration*

It should also be possible to only record changes to parts of the configuration. This can be useful when working with large and complex configurations, where only a small subset of the configuration is relevant.

FR 3 *Control when recording is active*

The user should have full control over when the recording is active. This means that the solution should offer an easy and intuitive way for the user to start and stop the recording process. It should also be possible to configure the solution to automatically start recording.

FR 4 *Apply the changes to other systems*

In order to put session recording to effective use, it must be possible to apply the performed changes on other systems too. This means that the solution should be able to generate configuration files or scripts that can be used to apply the changes to other systems. The solution should integrate with well-known and widely-used tools, for example, Ansible.

FR 5 *Apply a subset of the changes to other systems*

Not all systems are built equally. In an agile and iterative configuration process, one might start by building the configuration on a single system. After a working configuration has been found, it is likely that the single system will be split into multiple heterogeneous systems. Therefore it is a necessary requirement that the changes can be split and applied to the respective systems.

3.2 Non-Functional Requirements

Non-functional or quality requirements describe the quality attributes of the software. As pointed out by various scholars and practitioners, these requirements are at least as important factors for the success of any software as functional requirements are.[26, 27, 28].

NFR 1 *Transparency to the user*

The session recording feature should not disrupt the user's experience and workflow. The user should be able to use all tools and applications as usual without any noticeable difference, except when starting or stopping the recording.

NFR 2 *Transparency to the applications*

The application should not require any changes to support the session recording feature. There should be no impact on the application's performance, and no run-time checks should be necessary.

NFR 3 *Minimal overhead when not in use*

The session recording feature should have minimal overhead when not in use. The recording feature should not affect the system or application's performance, including processing time and memory usage.

NFR 4 *Concurrent process safety*

In a real-world system, multiple processes might modify the configuration concurrently. The solution should be universally applicable and be able to record all changes made by those concurrent processes reliably.

NFR 5 *Accuracy and completeness*

The system must be able to detect and record all changes made to the system configuration. There should be no false negatives, and all changes should be recorded accurately and completely.

CHAPTER 4

Approach

In Figure 4.1, we illustrate the overall workflow facilitated by our solution. An administrator iteratively modifies the configuration on a single machine. Once the configuration is finalized, these modifications are available in a reproducible form that is applied to other similar systems as well.

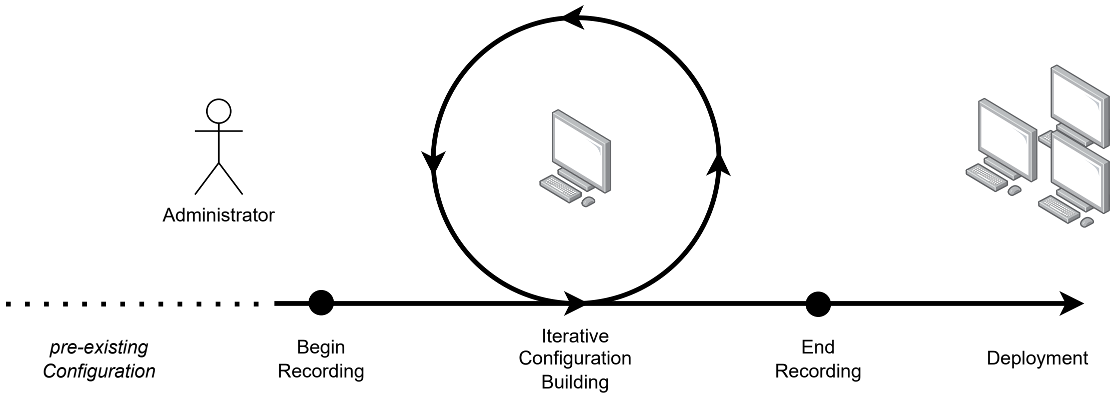

Administrator
pre-existing Configuration
Begin Recording
Iterative Configuration Building
End Recording
Deployment

Figure 4.1: Workflow overview of iterative configuration building

Our approach incorporates an intelligent monitoring system that automatically detects and records any modifications made to the system's configuration. This eliminates the need for the user to perform the configuration changes to manually keep track of the alterations. Instead, our solution offers a seamless and automated method for exporting and deploying these changes to other systems. By automating this process, we vastly reduce the administrative burden and enhance efficiency.

4.1 Tracking Configuration Changes using Elektra

We decided to implement the configuration tracking aspect of our approach using Elektra. This offers several distinct advantages. One notable benefit is the existing integration Elektra has with numerous applications. This pre-established integration simplifies the adoption process and ensures compatibility with a wide range of software systems.

Additionally, Elektra's mounting mechanism enables the creation of ad-hoc integrations effortlessly, without requiring modifications to the application's underlying code. This flexibility allows for seamless integration with various applications, expanding the versatility and adaptability of our approach.

Elektra provides two main functions for applications to interact with the configuration database. The *kdbGet* function retrieves configuration data from the key database, granting access to values linked to specific keys. On the other hand, the *kdbSet* function is utilized to store configuration data in the database, enabling modifications or additions to settings.

The essential concepts in our approach are denoted by the terms *part diff* and *session diff*, as depicted in Figure 4.2. A *part diff* captures the modifications made between a single *kdbGet* and *kdbSet* function call. On the other hand, a *session diff* encompasses all changes recorded during an active recording session. These terms serve as key components in understanding and analyzing the modifications made within our approach.

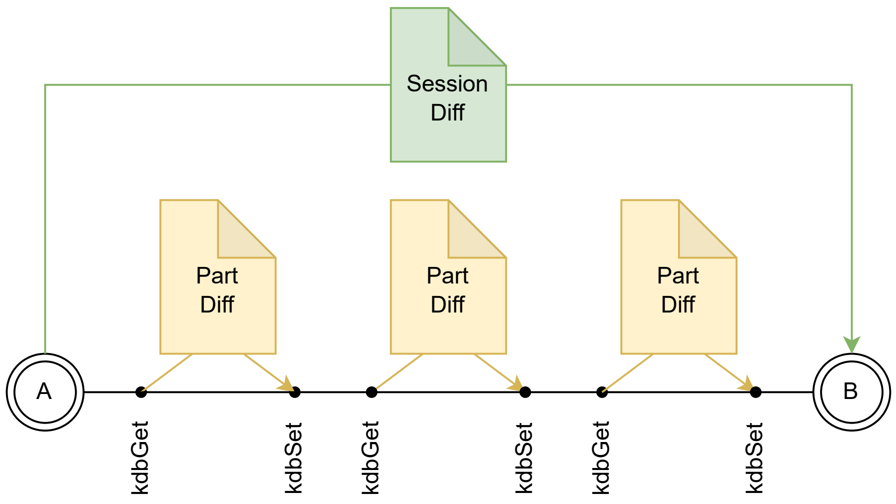


Figure 4.2: Aggregation of individual part diffs into a comprehensive session diff

Our general approach involves performing a deep duplication of the keyset obtained during the *kdbGet* operation. We then compare this duplicated keyset with the one provided in the *kdbSet* function. The outcome of this comparison yields the part diff, which represents the specific differences between the two keysets. By employing this approach, we can effectively identify and analyze the modifications made during the transition from reading the keyset to applying the changes in the *kdbSet* operation.

4.1.1 Calculating the session diff

Keys in a diff are divided into three distinct categories:

- *Added*: the key is new and did not exist before.
- *Modified*: the key existed before and still exists but its value or metadata has been modified.
- *Removed*: the key has been removed.

Keys that have not been modified and therefore are not represented in a diff are called *unchanged* keys in the following paragraphs.

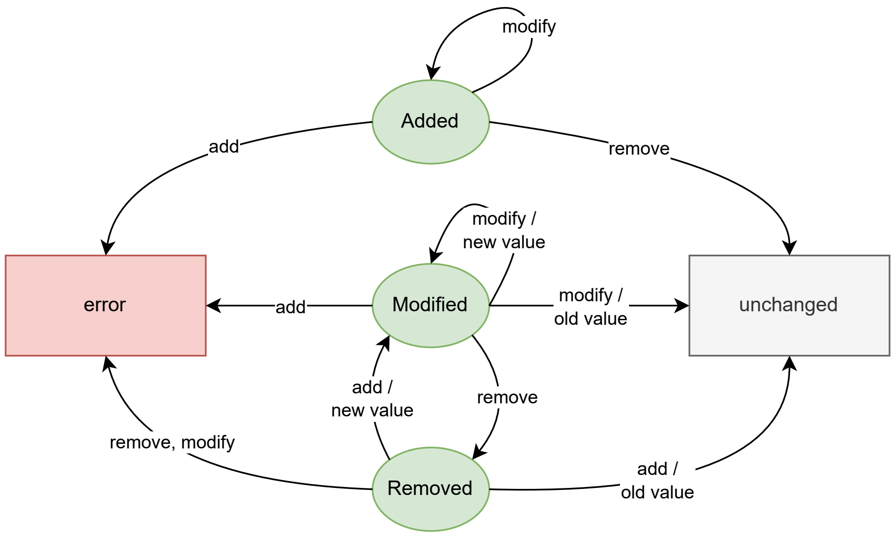


Figure 4.3: Key state transitions (without transitions from *unchanged*)

The diagram shown in Figure 4.3 visualises the state transitions when merging diffs. Green ovals depict the state of a key in the session diff. Arrows present the actions/state of a key in the new part diff.

For example, if a key is in *Added* state in the session diff, and it is in *Removed* state in the new part diff, then the key will be *unchanged* in the new version of the session diff.

4.1.2 Storage of the Session Diff

We made the decision to store the session diff directly within Elektra. This approach offers several advantages, including the ability to store all possible values of keys and supported metadata within Elektra. Moreover, leveraging Elektra's well-tested IO capabilities allowed us to fully utilize its robust infrastructure.

The solution specifically focuses on persistable namespaces, namely `dir:/`, `user:/`, `spec:/`, and `system:/`. For these namespaces, we store the modified keys within the respective namespaces themselves. To facilitate this, we mount a file in each namespace that serves as a container for the session recording data. The recorded data can be accessed under the path `/elektra/record/session` within each of the mentioned namespaces.

Here are the namespaces where the session recording data is stored:

- `dir:/elektra/record/session`
- `user:/elektra/record/session`
- `spec:/elektra/record/session`
- `system:/elektra/record/session`

This approach ensures that the recorded changes are organized and accessible within their respective namespaces, simplifying the management and retrieval of session recording data.

The following key and key hierarchies are important:

- `/elektra/record/config/active`: If this key is present, it signifies that session recording is enabled. The value of this key represents the parent key of the session. Only changes made to keys that are at the same level or below this parent key will be recorded.
- `/elektra/record/session`: Contains all the recorded data. It should be mounted into separate files in each namespace to ensure proper organization and management of the recorded sessions.
- `/elektra/record/session/diff/added`: Stores all the keys that have been added during the session.
- `/elektra/record/session/diff/modified/old`: Holds the previous values and metadata of the keys that have been modified during the session.
- `/elektra/record/session/diff/modified/new`: Stores the updated values and metadata of the keys that have been modified during the session.

- `/elektra/record/session/diff/removed`: Contains all the keys that have been removed during the session.

Whenever a key is added, removed, or modified, a corresponding clone of that key is created and stored in the appropriate location within the session storage. For instance, if a key `user:/test/name` with the value *Max* is added, a key `user:/elektra/record/session/diff/added/test/name` with the value *Max* is also generated and stored.

4.1.3 Hooks

Elektra has a highly flexible architecture that allows for easy extension through the use of plugins. As a matter of fact, most of the functionality of Elektra itself is implemented using plugins. Generally, plugins are bound to a specific mount point. So plugins act like steps in a pipeline.

In previous work, there was a concept known as *global plugins*, but it had several architectural flaws. One significant drawback was that these plugins were executed for every storage plugin, resulting in limited visibility of only certain parts of the configuration at any given time. Additionally, there were fixed phases for mounting plugins, a restriction on the number of global plugins allowed, and the limitation of using the same plugin interface as normal plugins.

Within the plugin system, we instead introduced a feature known as *hooks* to enhance its functionality. Hooks represent specific stages within the KDB lifecycle where specialized plugins, referred to as *hook plugins*, are invoked. Each hook has a well-defined API that outlines its behaviour and requirements. This way, we can present the plugins with a holistic view of the configuration. We can also ensure that they are only called once.

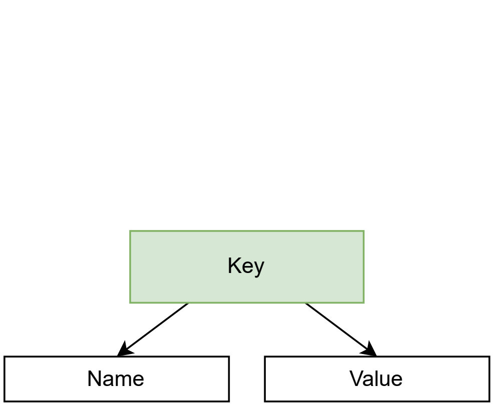
Our system incorporates multiple hooks, including the *notification* hook, which employs inter-process communication mechanisms to notify applications of configuration changes. Additionally, we have a dedicated *recording* hook designed specifically for session recording purposes, addressing the unique needs of that use case.

4.1.4 Copy-On-Write

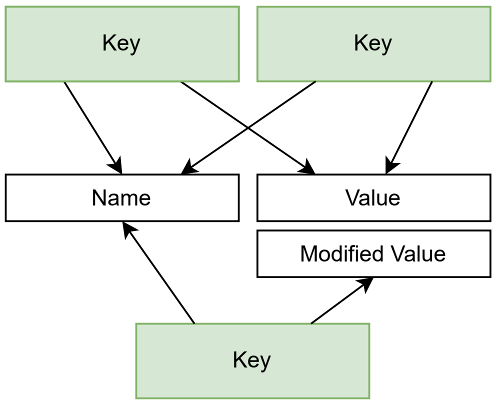
Our approach to tracking changes necessitates significant duplication of data. In order to minimize memory usage, we have opted for a copy-on-write strategy for the *Key* and *KeySet* data structures, which are widely utilized within Elektra.

When a key or keyset is copied or duplicated, only a shallow copy is made, containing references to the original data such as name, value, and contained keys. As the shared data is modified, new memory is allocated to preserve the integrity of the shared version. Consequently, duplicated keys and keysets consume only a fraction of the memory compared to their source counterparts.

The approach is depicted in Figures 4.4 and 4.5. Figure 4.4a depicts a single key. Figure 4.4b depicts three copies of a key, with one copy having been given another value.

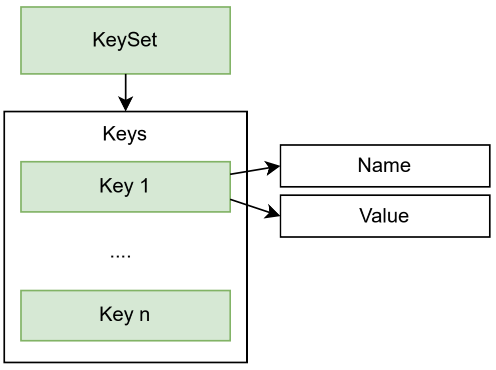


(a) A single key

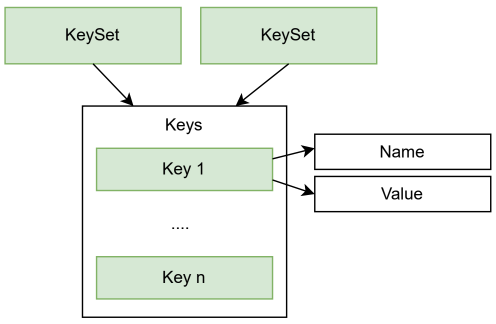


(b) Three duplicated keys

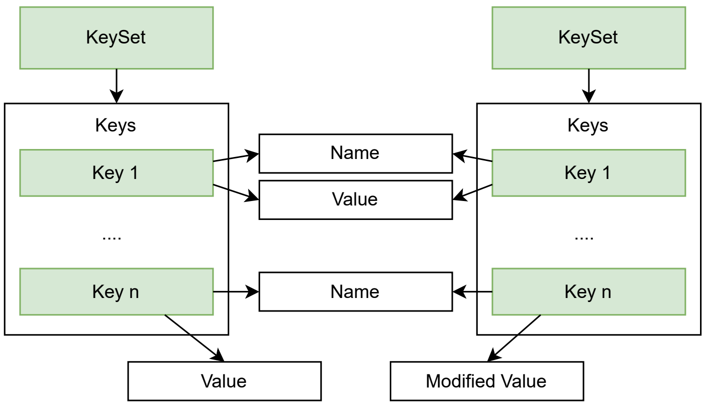
Figure 4.4: Copy-on-Write keys



(a) A single keyset



(b) Two duplicated keysets



(c) Two deep-duplicated keysets

Figure 4.5: Copy-on-Write keyset

Figure 4.5a illustrates a single keyset, which contains references to keys. The keyset itself does not store the actual key data. Figure 4.5b portrays a duplicated keyset where both keysets reference the same keys. Consequently, any modifications made to a key in one keyset will also affect the other keyset. When a key is added or removed from a keyset, it obtains its own list of keys. However, the keys they have in common remain shared.

Figure 4.5c showcases deep-duplicated keysets. Each keyset possesses its own list of keys, and those keys are duplicates as well.

4.2 Deploying Changes with Ansible

While Elektra is a powerful tool for managing configuration on a single machine, it does not have, nor strives to have, functionality to work across system boundaries. To enable the deployment phase of our workflow, we decided to use Ansible, which excels in automating tasks across systems, for rolling out the iteratively created configuration.

Two crucial components that facilitate this process include a custom Ansible module and an Elektra plugin capable of exporting Ansible playbooks. These exported playbooks can be utilized independently or integrated into broader and more comprehensive deployment scenarios.

4.2.1 Ansible Module

For a while now, there has been a basic Ansible module available for interacting with Elektra. However, this module had several limitations in terms of functionality. Therefore, we made the decision to completely overhaul and rewrite the module to incorporate all the necessary features required to support our desired workflow.

Transactionality

The revamped module now offers complete transactional support. In the event of an error occurring during task execution, all changes made are automatically rolled back, ensuring the integrity of the configuration.

Support for Multiple Namespaces

We have extended the module to allow for setting and modification of keys across multiple namespaces. Together with the transactionality feature, we can now ensure the atomicity of deploying configuration changes.

Removing Keys

The new module now allows for removing already existing keys. This functionality was not something that was previously of use outside of session recording. We can finally ensure that only certain keys are present on a machine.

Merge Strategies

Our updated module empowers us to perform three-way merges on the configuration of target machines. A three-way merge is a technique used to reconcile changes made to a dataset by multiple actors. It involves comparing the original (or *base*) version of the dataset with two modified versions, referred to as *theirs* and *ours*, to create a merged result that incorporates all the relevant modifications from both sources.

Additionally, we can define specific strategies to handle merge conflicts. The available strategies are:

- *Ours*: prioritize the changes made in the playbook,
- *Theirs*: prioritize the changes made on the machine,
- *Abort*: abort playbook execution and roll back all changes made to the configuration managed by Elektra during the execution of the task.

Control of Session Recording

The enhanced module provides control over the session recording feature. This enables the user to specify if session recording should be active on the machine, and which parts of the configuration should be monitored for changes. Additionally, it can be specified whether the changes performed during the playbook task execution itself shall be monitored.

4.2.2 Exporting Ansible Playbooks

Exporting the changes into an Ansible playbook is the final component needed to complete the puzzle. As Elektra is very extensible via plugins, we created a new Ansible export plugin.

While Elektra already possesses a robust exporting functionality, it falls short in a few critical aspects required for the task at hand. The most significant limitation is the absence of support for removed keys, which is a crucial concept for our use case. Additionally, Elektra lacks the ability to rewrite keynames, which becomes problematic as we are storing the changes under different names.

To address these limitations, we have developed a new export functionality specifically designed for session recording. Our goal was to make this export functionality as versatile as possible, allowing for the usage of existing export formats supported by the previous export functionality. Likewise, the export plugin that provides the Ansible playbook format can be utilized by the previous export functionality to export the entire machine configuration.

To overcome Elektra's lack of support for removed keys, we have introduced a special meta key named `meta:/elektra/removed` to serve as a marker for removed keys.

Plugins that explicitly support removed keys, such as the Ansible export plugin, can represent them in their native format. For all other plugins, the import tooling within Elektra may ensure that these keys are properly removed.

4.3 Tooling

Elektra has a wide variety of frontends that can interact with it. The two most important frontends for our approach are the aforementioned Ansible module and the *kdb* command-line tool. For both, we have created first-party integration with the session recording functionality. Other frontends are also easily adaptable, as we provide an easy-to-use library to administer session recording.

As for the recording itself, it is directly built into Elektra. Every application that uses Elektra in any form is automatically capable of recording changes to its configuration. Developers do not have to change a single line of code.

The command-line tool *kdb* now incorporates several commands specifically designed to interact with this feature. In order to clearly distinguish the commands that are specifically used for interacting with session recording, we made the decision to prepend the *record-* prefix to each of these commands. This prefix serves as a visual indicator for users, making it easily recognizable which commands are associated with session recording functionality.

kdb record-start [<parent_key>]
: This command initiates a session recording, capturing subsequent changes made to the configuration. The optional parameter *parent_key* can be used to limit the recording to a specific subtree of the configuration. If a previous session exists, only the parent key is modified.

kdb record-stop
: This command terminates the current recording session, preventing further changes from being recorded in the configuration. The recording session itself is not cleared and can be resumed using the *kdb record-start* command.

kdb record-export [<source>] [<format>]
: This command exports the recorded changes in the specified output format. The keys are exported to *stdout* in the specified format.

  The *source* parameter represents the path of the key(s) to be exported. Additionally, a specific format can be specified using the *format* argument. The *format* attribute relies on Elektra's plugin system to export the keys in the desired format. The *format* parameter must correspond to a valid storage plugin.

  Both *source* and *format* are optional parameters. By default, *source* is set to / and *format* is set to *ansible*. If only one argument is provided, the command determines whether it is a key or a format.

kdb record-state
This command displays information about the current state of the recording session. It includes whether recording is currently enabled, the future recording scope within the configuration (the session may already contain changes from other parts of the configuration), the number of recorded keys, and the specific keys that have been recorded.

kdb record-reset
This command clears all recorded changes in the configuration. However, it does not undo the actual changes made. The command can be executed regardless of whether recording is active or not.

kdb record-rm <key>
This command removes the specified *key* from the recording session, effectively excluding it from the recorded changes. If the key is modified again after being removed from the session, it will be included in the session once more.

kdb record-undo [<parent_key>]
This command undoes all the changes that have been recorded. The optional parameter *parent_key* can be used to limit the undo operation to a specific subtree of the configuration. After execution, the undone keys are no longer part of the recording session.

CHAPTER 5
Case Study

This chapter aims to answer Research Question 1.

Research Question 1 *In which way does Record Elektra benefit the use case of rolling out changes of configuration?*

We do so by demonstrating the capability of the Record Elektra tool to effectively manage any kind of configuration on a real-world embedded computer system.

5.1 Scenario

Opensesame[1], an open-source home automation software, is the primary focus of our real-world scenario for configuration management. The Opensesame stack will be run on an OLIMEX OLinuXino LIME 2[2] open-source single-board computer. We will also configure the */etc/hosts* and */etc/default/keyboard* files using Elektra. We chose this setup to investigate a medium-sized scenario, that contains both fully elektrified applications, as well as third-party integration for external configuration formats.

5.1.1 Scenario configuration overview

We are in the position of a corporation that has recently constructed a new office space. In accordance with building and employee safety regulations, each room must be monitored for both CO_2 concentration and smoke accumulation. If either of these values surpasses a specified threshold, an alarm will be activated within the room, and a notification will be sent via the company's Nextcloud chat service.

[1] https://github.com/elektraInitiative/opensesame
[2] https://www.olimex.com/Products/OLinuXino/A20/A20-OLinuXino-LIME2/open-source-hardware

To ensure redundancy in the system, the company has opted to install a single breadboard computer equipped with sensors in each room. Each of these computers runs on the Linux operating system with Opensesame.

5.1.2 Iterations

Since this is an unfamiliar area for the IT staff, their plan is to gradually construct the configuration on a single machine. Once they have completed this process, they aim to implement the same configuration on all other systems as well. This enables the IT staff to test the configuration and identify any potential issues or bugs before deploying it across all other machines.

Iteration 1 - Base Settings

As a first step, all basic settings of the system shall be configured. It encompasses adding an entry for the Nextcloud server to */etc/hosts*.

To maintain optimal security, it is imperative to establish separate login credentials for the Nextcloud instance of each company host. Consequently, the configuration for each host must be customized in a unique manner. Having distinct login credentials for each host enables the system administrators to more easily identify and trace any security breaches or malicious activity.

The sensors for the room also need to be configured in this step. For this, we have pre-determined suitable values for the triggering of the alarm and the chat notification for both the CO_2 and smoke sensors. Furthermore, we have to configure which audio file is played for the alarm.

After testing the configuration on a single host, the configuration is deemed okay and ready to be rolled out to every host.

Iteration 2 - Configure Settings That Only Apply to Certain Hosts

It was decided that in common areas, like the kitchen, the computers will also feature an interactive terminal. The idea is to let users view the current and historical air quality. On those hosts, we will need to configure the keyboard layout.

Iteration 3 - Needless Fine Adjustments

Following a period of time, it is decided to make some minor tweaks to the sensor configurations. However, after conducting experiments and comparing results with other hosts, it was determined that the original configuration was actually the most effective. Therefore, we must now reverse all the modifications made on this particular host.

5.2 Preperatory Steps

Before proceeding with any configurations, there are several steps that need to be completed manually, without utilizing Elektra. Initially, it is necessary to download the operating system[3] and then flash it onto a microSD card of the appropriate size. This particular step cannot be automated.

Once that is done, we can begin implementing automation. The system image already includes an SSH server that is pre-installed and enabled by default. To support Ansible, Python must be installed first. Fortunately, we can fully automate the installation of Python, as well as other software and dependencies, using Ansible.

For our specific case study, there are two main components that need to be installed: Opensame and Elektra. Since the software has not been officially released at the time of writing this thesis, we had to create a customized Debian repository. This repository needs to be registered on the system, and the software can then be installed from there. Detailed instructions and playbooks for these preparatory steps, along with the overall container playbook, can be found on the accompanying website for this thesis - `https://thesis.maxirlinger.at/#downloads`.

5.3 Generating the Configuration

Once all prerequisites are installed, we can establish a connection to the first host and commence the configuration process. The following subsections will list all the commands that will be executed on the host during the configuration process, as well as the used configuration data. The initial step involves enabling session recording.

```
kdb record-start /
```

5.3.1 Base Settings

To begin, we need to configure the */etc/hosts* file. For this purpose, we must first mount it into Elektra.

```
kdb mount -W /etc/hosts system:/hosts hosts
```

Next, we set the appropriate entry for our Nextcloud instance in the hosts file. It is important to note that Elektra validates whether the provided input is a valid IP address, which helps eliminate a wide range of potential errors.

```
kdb set system:/hosts/ipv4/nextcloud.maxirlinger.at 65.108.157.177
```

[3]http://images.olimex.com/release/a20/

Now we can proceed to configuring Opensesame. We have the option to either supply a dedicated *kdb* command for each key, as we did for the hosts file, or utilize Elektra's built-in editor functionality. This feature enables administrators to use their preferred text editor to modify the configuration in a specified format. Once the file is saved and the editor process is closed, Elektra will re-import the file and perform checks to verify the validity of the configuration. If any errors are detected, an error message will be displayed and the configuration will not be applied.

```
kdb editor system:/sw/libelektra/opensesame/#0/current toml
```

At this point, we can provide our initial set of Opensesame settings in TOML format. We begin by supplying the Nextcloud credentials, as they are required for Opensesame to start up. Sensitive information in the configuration has been replaced with ⋆⋆⋆⋆.

```
[nextcloud]
url = "https://nextcloud.maxirlinger.at/"
chat = "⋆⋆⋆⋆"
chat.licht = "⋆⋆⋆⋆"
chat.ping = "⋆⋆⋆⋆"
user = "⋆⋆⋆⋆"
pass = "⋆⋆⋆⋆"
```

Opensesame will now raise an issue indicating that no sensors have been configured yet. We once again open the configuration file using `kdb editor` and add the following section at the bottom of the file.

```
[environment]
name = "corporate"

[[sensors]]
loc = "CO2 sensor"
alarm = 250
chat = 200

[[sensors]]
loc = "Smoke sensor"
alarm = 100
chat = 50
```

We have now registered two sensors with Opensesame. The first one is the CO_2 sensor, which will log an entry into Nextcloud when its reported value exceeds 200 and trigger an alarm when it exceeds 250. The second sensor is a smoke detector, configured in a similar manner. The values represent mappings from an analog pin, ranging from 0 to 3.3 volts.

Next, we can configure the audio file that should be played when the alarm threshold is reached.

```
[audio]
alarm = "/home/olimex/alarm.ogg"
```

This concludes the Opensesame-specific part of the configuration. Elektra automatically restarts Opensesame every time a configuration change is made. Now, we can export our Ansible playbook for the shared portion of the configuration.

```
kdb record-export ansible
```

It is important to highlight that we must make a manual adjustment in the Ansible file, replacing the Nextcloud credentials with Ansible variables. This approach allows each host to have its own unique set of credentials. Although the Ansible export plugin in Elektra can accommodate this requirement, the TOML storage plugin utilized by Opensesame lacks the capability to store the essential metadata for this purpose.

The recorded configuration will now be rolled out to all hosts that have been defined in our Ansible inventory file, using the *ansible-playbook* utility.

5.3.2 Settings That Only Apply to Certain Hosts

We proceed with configuring the machines that require a keyboard layout adjustment. To accomplish this, we login remotely to the respective host. Subsequently, we need to mount the */etc/default/keyboard* file in Elektra, allowing us to utilize the KDB editor once again for making the necessary changes. We do not need to activate session recording manually this time, as it has already been enabled by the deployment of the shared configuration via Ansible.

```
kdb mount /etc/default/keyboard system:/keyboard mini
kdb editor system:/keyboard mini
```

```
BACKSPACE="guess"
XKBLAYOUT="de"
XKBMODEL="pc105"
XKBOPTIONS="lv3:ralt_switch"
XKBVARIANT=""
```

All that's left is again to export theses changes. We get a new Ansible playbook, where we can specify which group of hosts from our inventory it will be executed on.

```
kdb record-export ansible
```

5.3.3 Needless Fine Adjustments

As mentioned earlier, if Ansible was used to configure the system, Elektra's session recording gets activated automatically. So when we experiment around with the sensor values, everything is logged. At the end of the day, we can just issue the undo command to get back to the previous state.

```
kdb record-undo
```

This concludes our configuration scenario.

5.4 Discussion

Based on the execution scenario discussed earlier, it can be concluded that Record Elektra is capable of effectively configuring and implementing iterative changes in configurations.

For the configuration on the host, we had to execute 10 commands, 5 of them were caused by configuring with Elektra. Two commands were the mounting of the */etc/hosts* and */etc/default/keyboard* files, and one command was the initial session recording start. Another two commands were issued to export the playbooks for the first two iterations.

The remaining 5 commands would also have been necessary, had we configured the system manually without Elektra. One command for adding the entry in the hosts file, one command for editing the keyboards file, and 3 commands for iteratively creating the configuration for Opensesame.

Additionally, we had to execute two commands on the managing computer, to roll out the Ansible playbooks. The generated playbooks for both iterations can be found on `https://thesis.maxirlinger.at/#downloads`.

With only a handful of commands more, and basically the same workflow on a single host, we have identified the following key advantages of our approach:

1. Effortless commissioning of new hosts with the same general configuration is made possible through our integrated automation using Ansible playbooks.

2. The built-in undo feature allows for safe experimentation on the configuration. Users can always restore the original, unmodified configuration with a single command.

3. Elektra includes built-in support for validating the correctness of configuration values. This ensures that we cannot assign incorrect values, such as garbage values, to IP addresses in the hosts file. Additionally, it provides immediate feedback if we attempt to configure values for the sensors with incompatible data types. Elektra could also verify the existence of the specified audio file for the alarm.

4. Elektra's ability to restart services upon configuration changes is a major benefit. Elektra successfully restarted Opensesame as needed.

Nonetheless, we have also identified a limitation in our solution. Working with host-specific values, such as the Nextcloud login credentials in our scenario, is generally feasible with our implementation. However, the approach we employed, utilizing metadata, is incompatible with certain storage plugins offered by Elektra. In the case of the TOML storage plugin, it was unable to store this metadata, resulting in the need for manual editing of the exported Ansible playbooks.

As for rolling out the changes onto a new host via Ansible, the completely automated procedure takes about 10 minutes. Of those 10 minutes, 1 minute is used to install Python in order for Ansible to work properly. Another 9.5 minutes are spent on installing Elektra and Opensesame. The remaining 30 seconds are the time it takes out to roll out the configuration itself.

CHAPTER 6

Benchmarking

The purpose of this chapter is to address Research Question 2.

Research Question 2 *What performance and memory impact does Record Elektra have compared to vanilla Elektra?*

This research question comprises two key components:

1. performance impact, and
2. memory consumption.

To determine the impact on both components, we examine the following four variants of Elektra:

1. pre-implementation (git hash 59d56a82a3)[1],
2. post copy-on-write implementation (git hash 1c7855fc50)[2],
3. post record-implementation (git hash 5746f8b56d)[3] with recording inactive, and
4. post record-implementation with recording active.

For these benchmarks, we created a benchmark program (located at *benchmarks/change-tracking.c* in Elektra's source tree) using Elektra's existing benchmarking framework. This benchmark consists of two rounds. We measure the time for the call of *kdbSet* each round with microsecond precision.

[1]https://github.com/elektraInitiative/libelektra/tree/59d56a82a3
[2]https://github.com/elektraInitiative/libelektra/tree/1c7855fc50
[3]https://github.com/elektraInitiative/libelektra/tree/5746f8b56d

1. Create a specified number of keys, each with the key name `user:/test/sw/org/myapp/#0/current/section/subsection/keyNUMBER`. Every created key gets a different value `valueNUMBER`.

2. Modify half of the created keys by giving them the value `value-modifiedNUMBER`.

To provide a thorough analysis of each impact, each benchmark was conducted with a series of key numbers ranging from 10 to 1,000,000 keys. The number of keys will be increased 10-fold each time. So the series consists of 10, 10^2, 10^3, 10^4, 10^5 and 10^6 keys. By doing so, we aim to gain a better understanding of the impact of our changes to Elektra on performance and memory consumption across different configurations and data sizes.

All benchmarks were carried out on the following reference system, a virtual machine on a Proxmox Server:

- CPU: 4 x Intel(R) Xeon(R) CPU X3450 @ 2.67GHz
- RAM: 8 GB DDR-3 1333 MT/s
- OS: Debian 11.6
- Storage: SAS 6G attached RAID 0 SSDs

As for the configuration of Elektra itself, we use a clean installation with the default settings and plugins. The following build flags were used to build each of the variants:

```
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo \
    -DKDB_DB_HOME="~/elektra-new/home" \
    -DKDB_DB_SYSTEM="~/elektra-new/system" \
    -DKDB_DB_SPEC="~/elektra-new/spec" \
    -DKDB_DB_USER="elektra-new/user" \
    -DPLUGINS=ALL \
    -DBINDINGS=ALL \
    -DCMAKE_INSTALL_PREFIX=/opt/elektra/
```

6.1 Threats to Validity

There was other development done on Elektra as well. Since only the "recording active" and "recording inactive" variants use the same git commit, differences between other versions may in part be influenced by changes other than the ones described in this thesis.

6.2 Performance Impact

The first impact we are investigating is the potential performance impact caused by our changes. Our primary focus lies on the total runtime of the implementation.

6.2.1 Methodology

We use a simple clock-on-the-wall benchmarking approach. The benchmark program reports its running time with microsecond-level accuracy. To ensure the reliability of the measurements, each benchmark is run ten times. We then calculate the median of the measured runtimes to ensure single outliers don't skew the result.

6.2.2 Results - Runtime for Creating Keys

In this section, we are presenting the variations in runtime when generating keys. The measured times for each variant and number of keys are presented in Table 6.1, providing a numerical summary. Figure 6.1 offers a graphical representation of how the median runtimes change with key size. Furthermore, Figures 6.2 to 6.7 present boxplots that visually illustrate the runtime, offering a more comprehensive understanding of the statistical variance in the measurements.

Variant / Key Count	10	100	1 000	10 000	100 000	1 000 000
Original	8.02	9.07	25.88	183.25	1874.77	18616.85
Copy-on-Write	9.45	11.08	26.16	183.49	1874.09	18169.58
Recording (inactive)	8.46	9.66	25.27	179.73	1804.28	17444.58
Recording (active)	18.91	21.55	59.38	430.01	4476.13	46162.38

Table 6.1: Runtime for creating for different variants in milliseconds

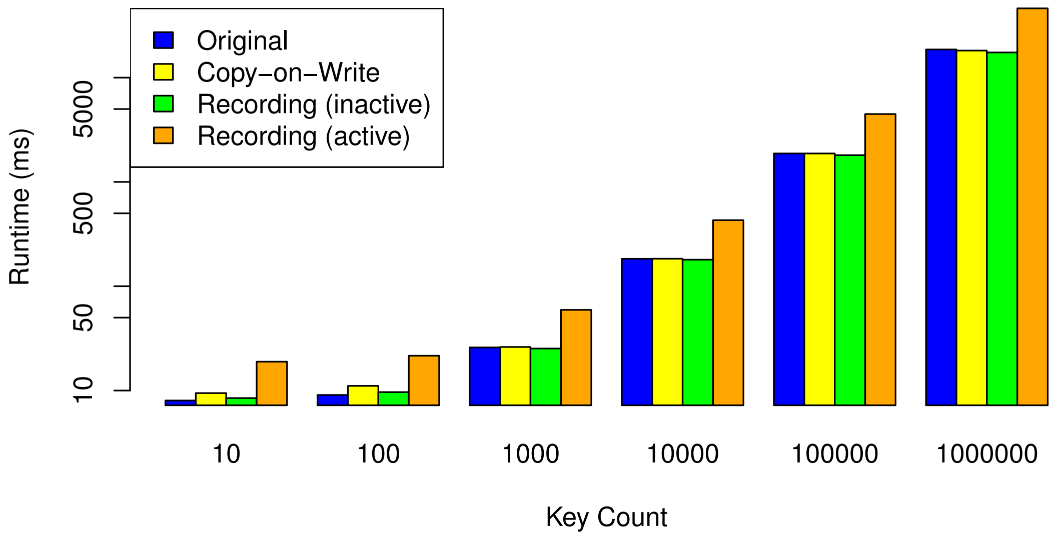


Figure 6.1: Comparison of median runtimes for creating keys

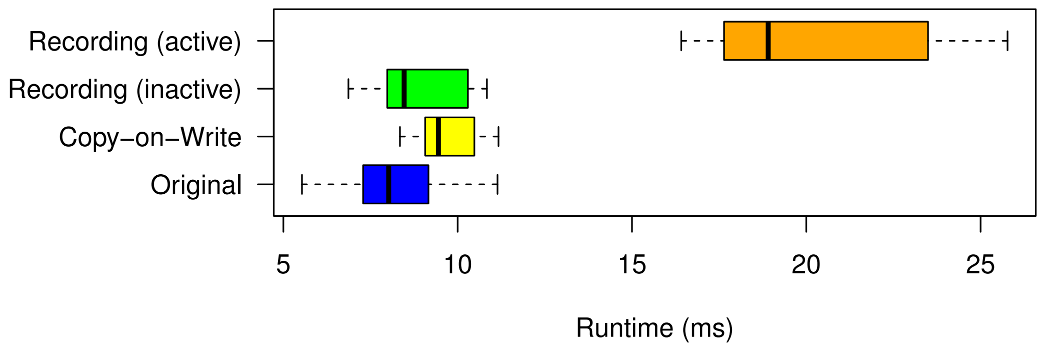


Figure 6.2: Total runtimes for creating 10 keys

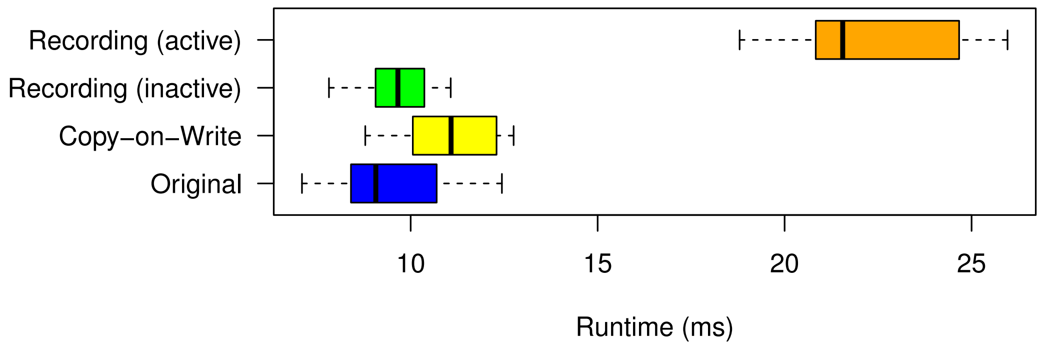


Figure 6.3: Total runtimes for creating 100 keys

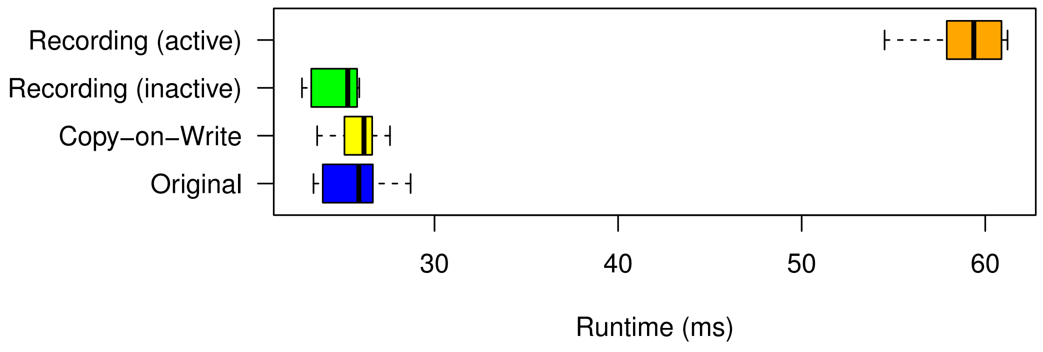


Figure 6.4: Total runtimes for creating 1 000 keys

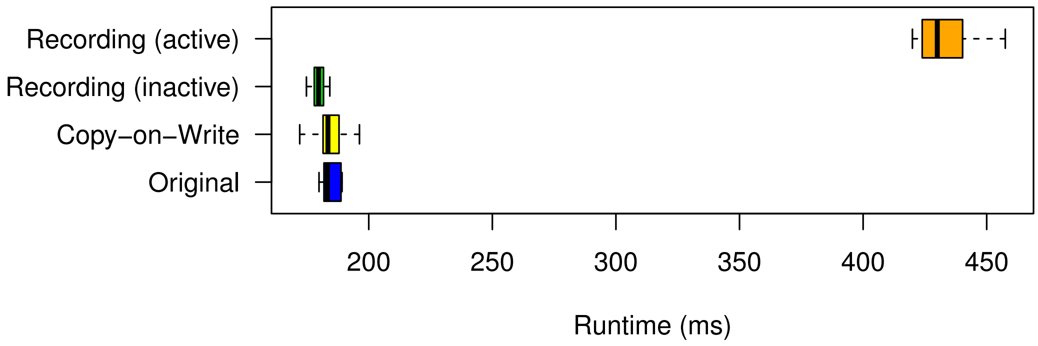


Figure 6.5: Total runtimes for creating 10 000 keys

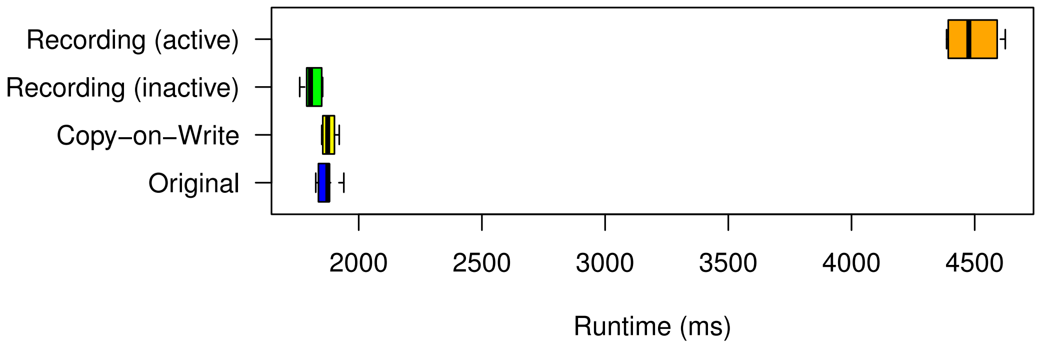


Figure 6.6: Total runtimes for creating 100 000 keys

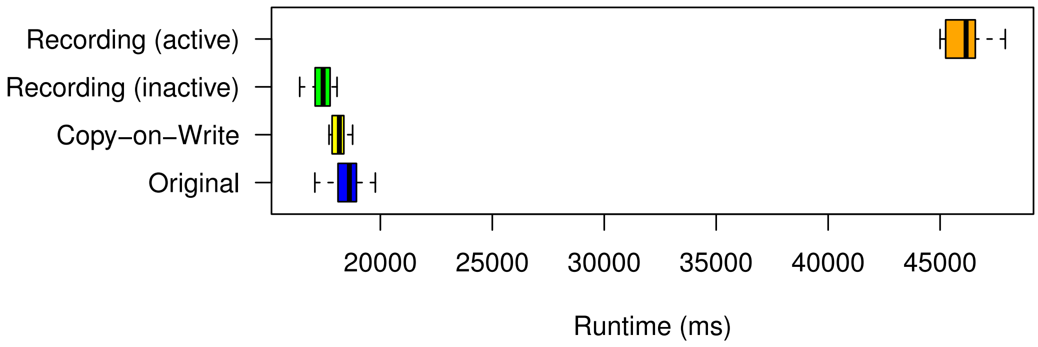


Figure 6.7: Total runtimes for creating 1 000 000 keys

6.2.3 Results - Runtime for Updating Keys

In this section, our focus is on examining the runtime progression during the process of updating keys. As mentioned earlier, half of the previously generated keys are assigned new values. The numerical summary of the runtime data can be found in Table 6.2. To provide a more graphical overview, the data is visually summarized in Figure 6.8. Additionally, Figures 6.9 to 6.14 display boxplots that visually depict the runtime, enabling a more comprehensive understanding of the statistical variance in the measurements.

Variant / Key Count	10	100	1 000	10 000	100 000	1 000 000
Original	7.95	9.07	24.02	181.41	1823.71	18389.56
Copy-on-Write	10.25	11.15	25.72	181.45	1835.95	18203.52
Recording (inactive)	7.9	9.54	24.81	178.43	1804.4	17805.12
Recording (active)	17.93	22.42	57.71	416.13	4330.15	44792.97

Table 6.2: Runtime for updating for different variants in milliseconds

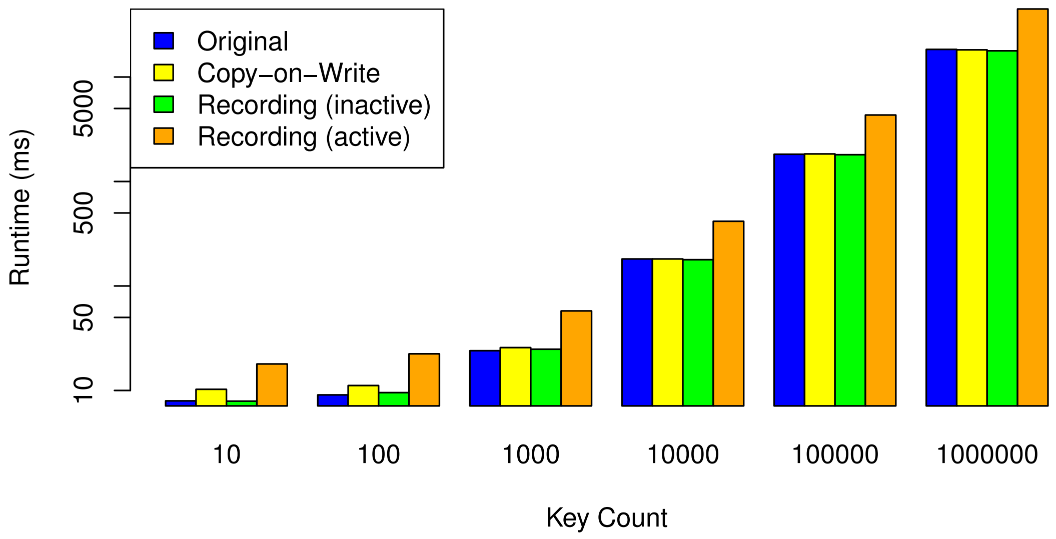


Figure 6.8: Comparison of median runtimes for updating keys

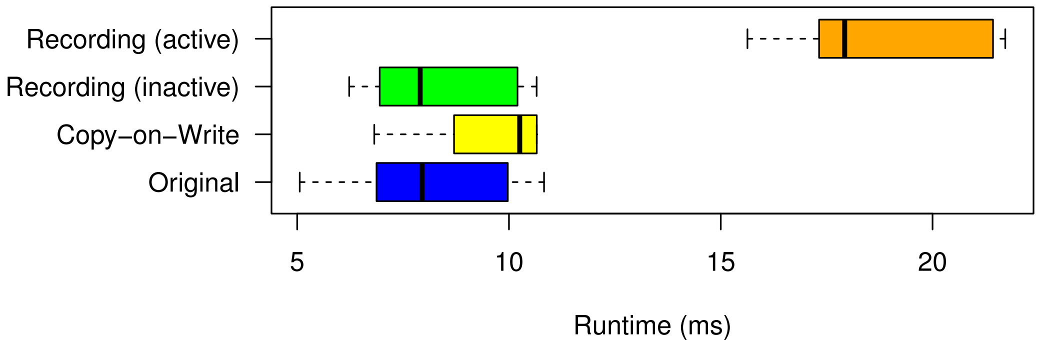


Figure 6.9: Total runtimes for updating half of 10 keys

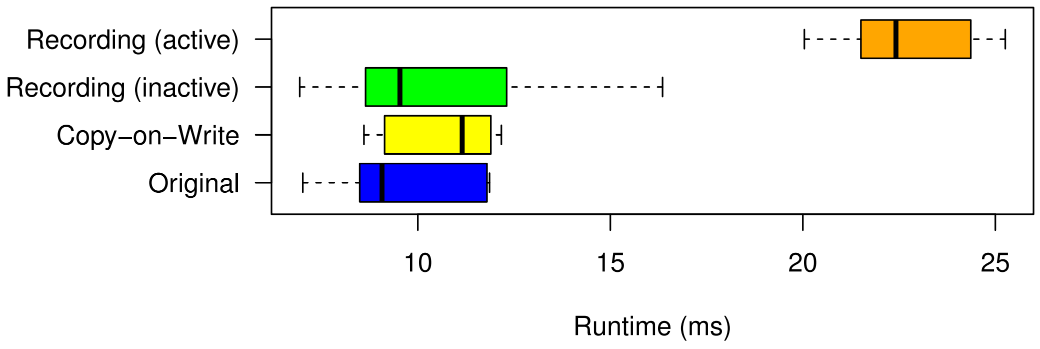


Figure 6.10: Total runtimes for updating half of 100 keys

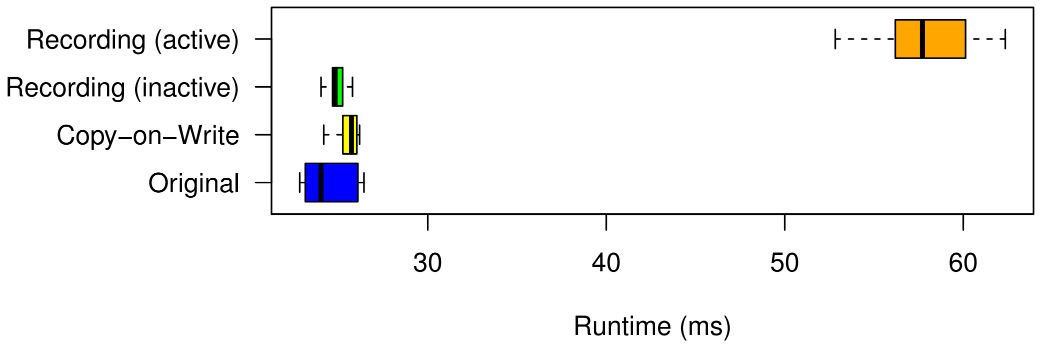


Figure 6.11: Total runtimes for updating half of 1 000 keys

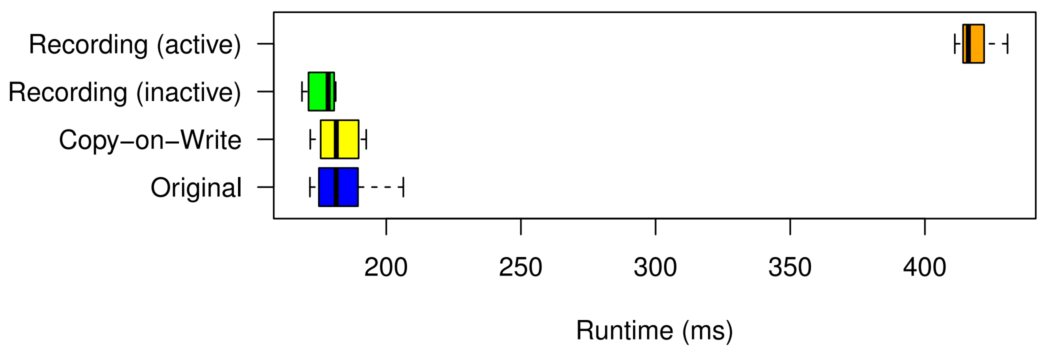


Figure 6.12: Total runtimes for updating half of 10 000 keys

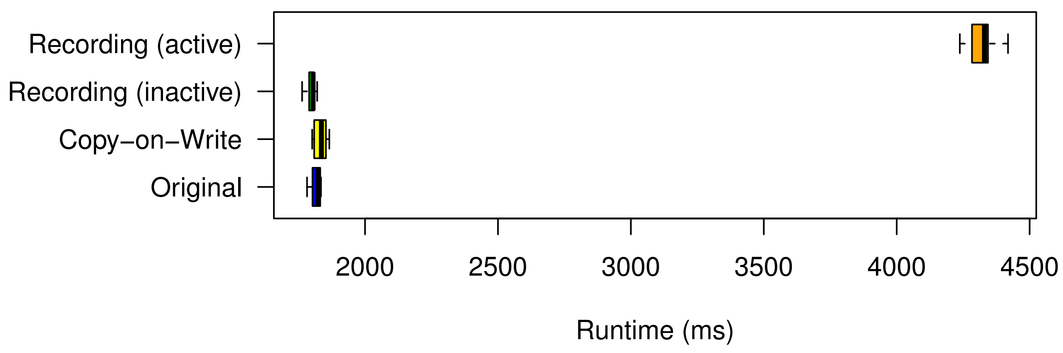


Figure 6.13: Total runtimes for updating half of 100 000 keys

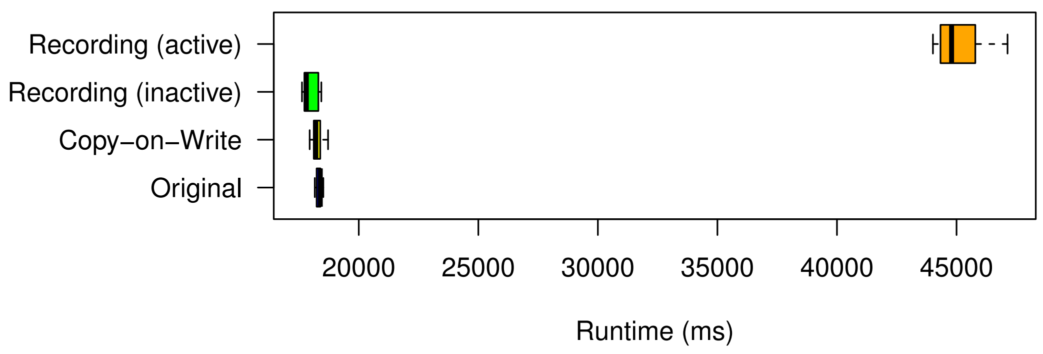


Figure 6.14: Total runtimes for updating half of 1 000 000 keys

6.3 Memory Consumption

We now turn our focus onto the development of the memory consumption of our changes.

6.3.1 Methodology

We utilize Valgrind's Massif heap profiler tool to track peak allocated memory in order to measure memory consumption. This tool is a well-established and reliable option for measuring memory usage in software implementations [29, 30].

6.3.2 Results

Similar to the preceding sections, we begin by presenting a numerical summary in Table 6.3 to showcase the memory usage. This information is further visualized in Figure 6.15, where we have plotted the memory consumption for each variant at different key counts. To delve deeper into the memory consumption analysis, Figures 6.16 to 6.21 provide a more detailed perspective on the memory usage for each variant.

Variant / Key Count	10	100	1 000	10 000	100 000	1 000 000
Original	0.18	0.23	0.81	6.8	66.82	669.47
Copy-on-Write	0.18	0.21	0.51	3.64	34.55	340.98
Recording (inactive)	0.19	0.23	0.58	4.25	40.28	396.63
Recording (active)	0.29	0.39	1.42	12.29	119	1174.94

Table 6.3: Peak memory consumption with different key counts in Mebibytes (MiB)

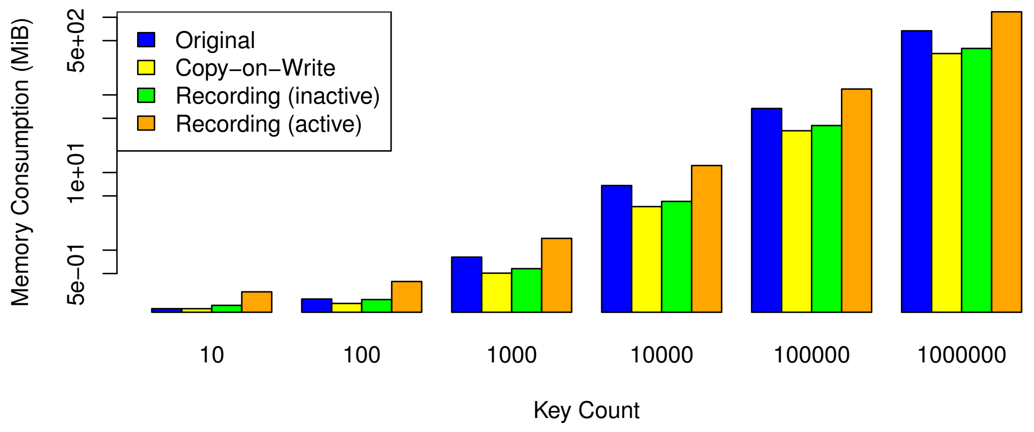


Figure 6.15: Peak memory usage comparison

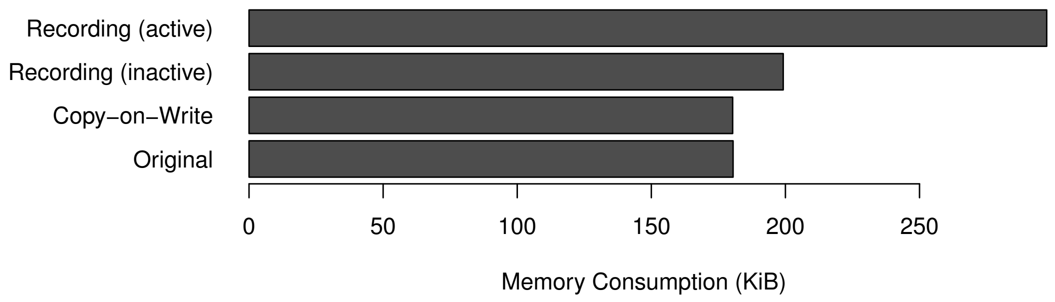


Figure 6.16: Peak memory consumption with 10 keys

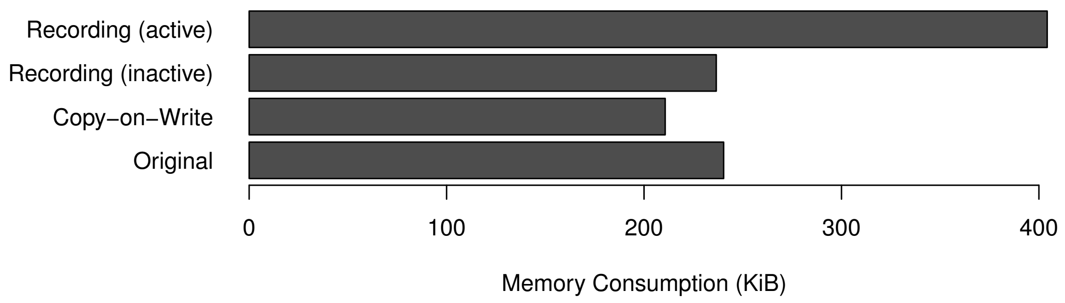


Figure 6.17: Peak memory consumption with 100 keys

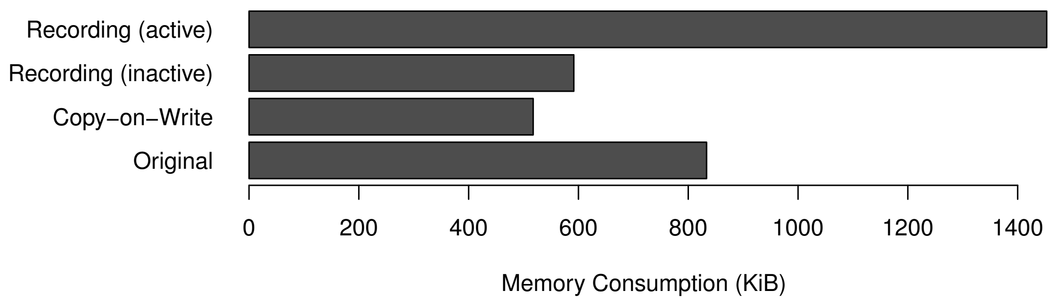


Figure 6.18: Peak memory consumption with 1 000 keys

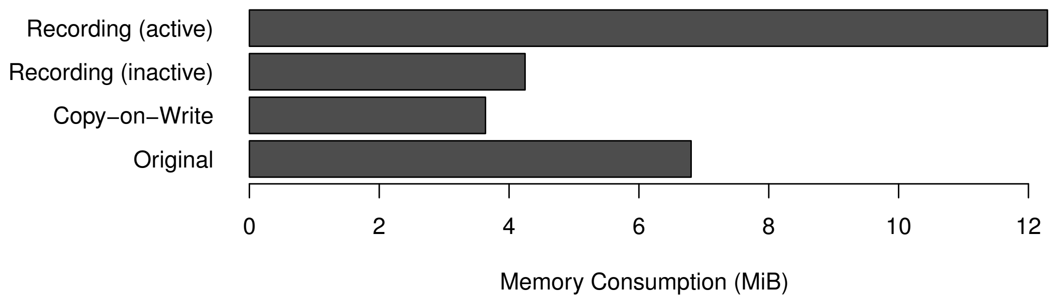


Figure 6.19: Peak memory consumption with 10 000 keys

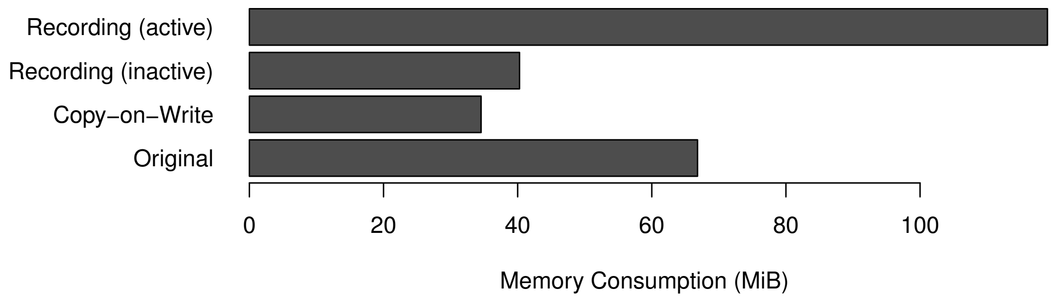


Figure 6.20: Peak memory consumption with 100 000 keys

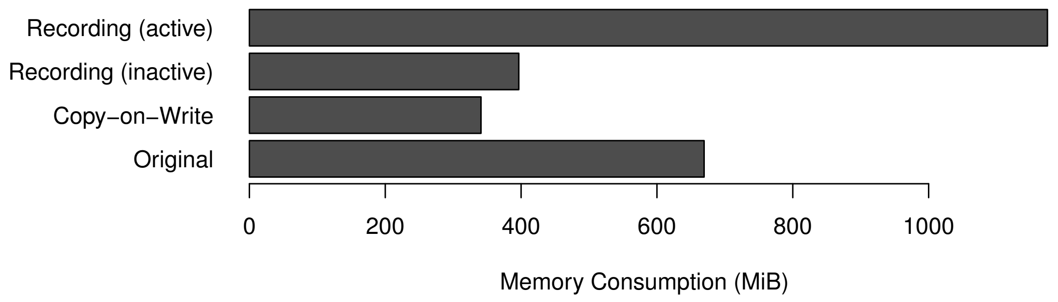


Figure 6.21: Peak memory consumption with 1 000 000 keys

6.4 Callgraph Analysis

Before we discuss the raw data presented in the previous sections, we want to gain a bit more insight into where the runtime overhead comes from.

6.4.1 Methodology

To identify the source of performance bottlenecks, Valgrind's Callgrind [31, 32, 33] tool is employed. This tool has been extensively used in prior research and has been shown to provide accurate and detailed information on the execution of software.

We concentrate on analyzing the callgrind output for 1 million keys with recording active. Our specific focus is on identifying the top functions that contribute to the slowdown when recording is enabled. To facilitate this analysis, we have simplified the provided callgraph. In these diagrams, we have highlighted the functions specific to session recording as green boxes.

6.4.2 Analysis

Initially, we concentrate on examining the *kdbSet* function, which is called by the benchmark program. This particular callgraph is presented in Figure 6.22.

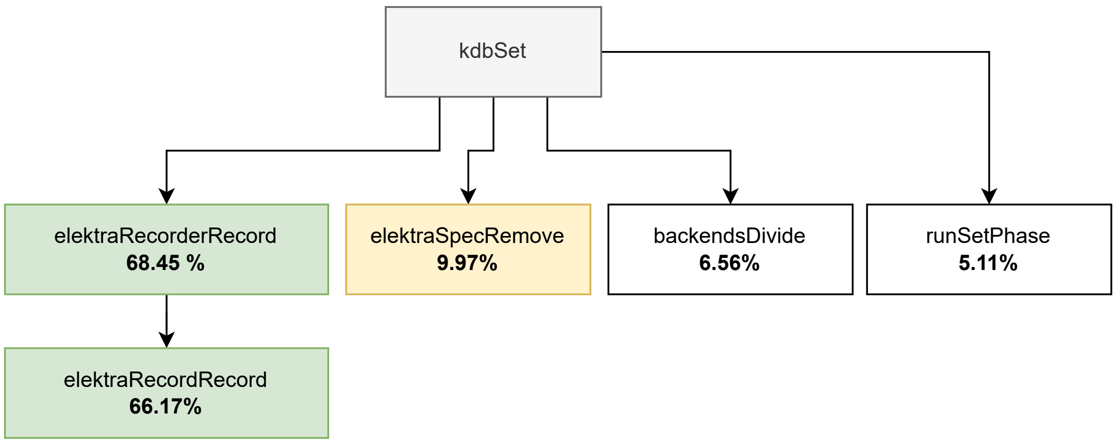


Figure 6.22: Callgraph part for main *kdbSet*

As we see in Figure 6.22, recording makes up 68.45% of the total time spent in *kdbSet*. The function *elektraRecorderRecord* is the main entry point of the recording hook plugin, whereas *elektraRecordRecord* is the function that is actually responsible for recording.

Following that, we come across another time-consuming operation, accounting for 9.97% of the overall execution time, known as *elektraSpecRemove*. This function is responsible for handling the cleanup of specification data and acts as the entry point for a hook, specifically the *specification* hook.

The remaining prominent functions in terms of execution time are *backendsDivide*, which occupies 6.56% of the execution time, and *runSetPhase*, accounting for 5.11%. These functions primarily focus on persisting data and executing all non-hook plugins specified within the configuration.

As we are mainly concerned with analyzing the overhead of recording, we jump right into the sub-graph for *elektraRecordRecord* in Figure 6.23.

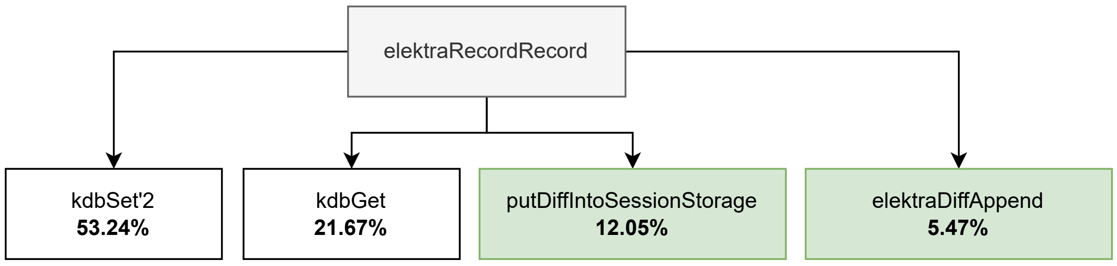


Figure 6.23: Callgraph part for *elektraRecordRecord*

As previously stated, we employ Elektra for storing the recording data. After a short glance, it becomes apparent that the combined execution time for invocations of *kdbGet* and *kdbSet* accounts for 74.91% of our overall runtime. To distinguish the original *kdbSet* call from the one within *elektraRecordRecord*, we have labelled the latter as *kdbSet'2*.

The remaining two significant functions, *putDiffIntoSessionStorage* and *elektraDiffAppend*, contribute 12.05% and 5.47% respectively to the overall execution time. The function *elektraDiffAppend* is responsible for implementing key state transitions, as illustrated in Figure 4.3, and thus plays a crucial role in executing a major part of the recording logic.

To complete our deep-dive, we will take a final look at the callgraph for *kdbSet'2* in Figure 6.24.

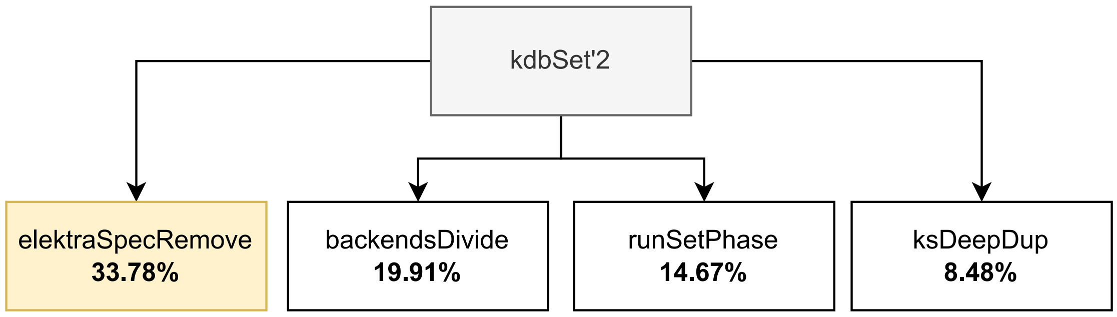


Figure 6.24: Callgraph part for secondary *kdbSet*

As this function call does not execute any recording-specific functions, we gain a clearer understanding of the performance bottlenecks within *kdbSet*. Highlighted in yellow, we can once again observe the significant contribution of the *elektraSpecRemove* function, which accounts for 33.78% of the total runtime this time.

The functions *backendsDivide* and *runSetPhase*, which contribute 19.91% and 14.67% respectively, are also already familiar. The only new function here is *ksDeepDup* with

8.48%, which is used within multiple places in *kdbSet* to create deep-duplications of keysets.

6.5 Discussion

From the runtime observations, the first notable conclusion is that when recording is disabled, our changes have minimal impact on performance. In fact, thanks to our optimizations we implemented in Elektra, the runtime performance is even better than before for key counts of 1000 and above. For smaller key counts, the performance remains nearly identical. Furthermore, we have observed that the performance characteristics are comparable between updating and creating keys.

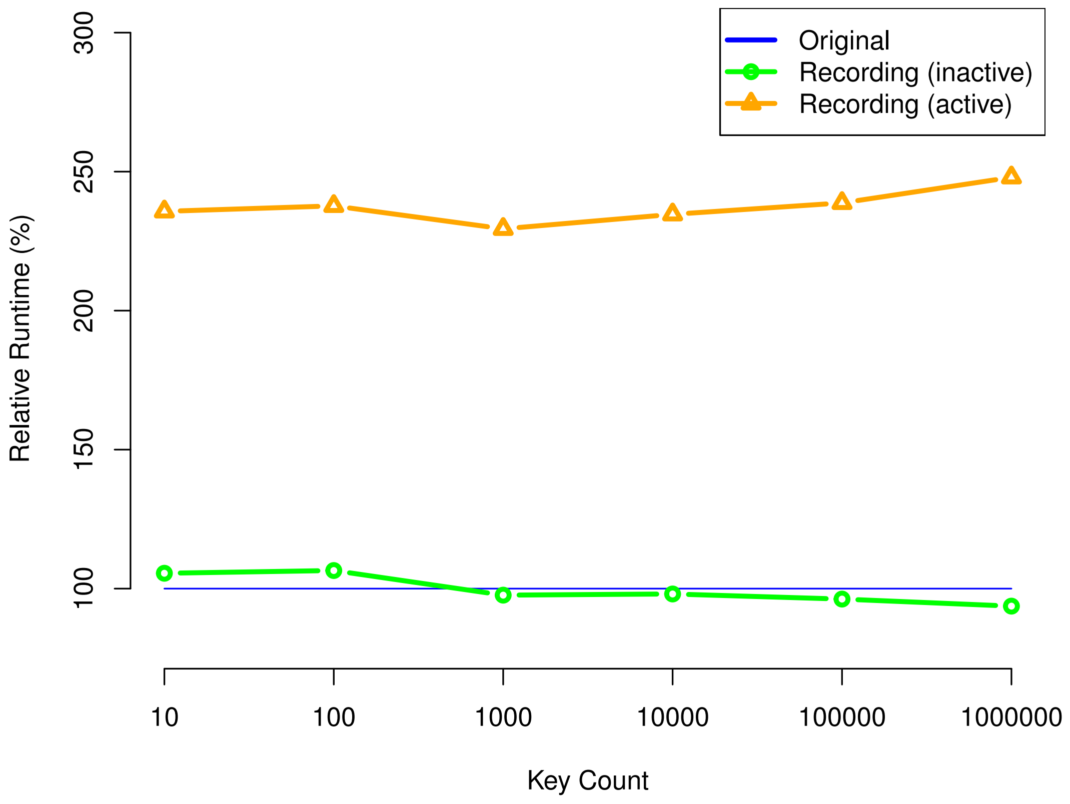


Figure 6.25: Runtime impact of our changes relative to the original implementation

Enabling recording introduces a noticeable performance impact, resulting in approximately 2.5 times increase in runtime compared to when recording is inactive. This is clearly depicted in Figure 6.25, where the relative performance of creating keys with recording active and inactive is compared to the original implementation, providing a visual representation of the impact.

In Section 6.4, we conducted a detailed analysis of the callgraph for session recording. The primary observation was that session recording accounts for approximately two-thirds of the total runtime of the *kdbSet* function. This finding aligns with the 2.5 times increase in runtime observed in the time-based benchmarks.

Furthermore, we found that nearly three-quarters of the session recording overhead can be attributed to the operations of reading and persisting the session diff using the *kdbGet* and *kdbSet* functions. This insight suggests clear optimization opportunities, such as disabling the *spec* hook when using Elektra as record storage since it is unnecessary in this context. Other areas of possible future optimizations are building keysets and renaming keys.

Regarding memory consumption, we can draw parallel conclusions. We have represented the relative memory consumption in Figure 6.26, demonstrating the comparison between active and inactive recording, as well as the original implementation.

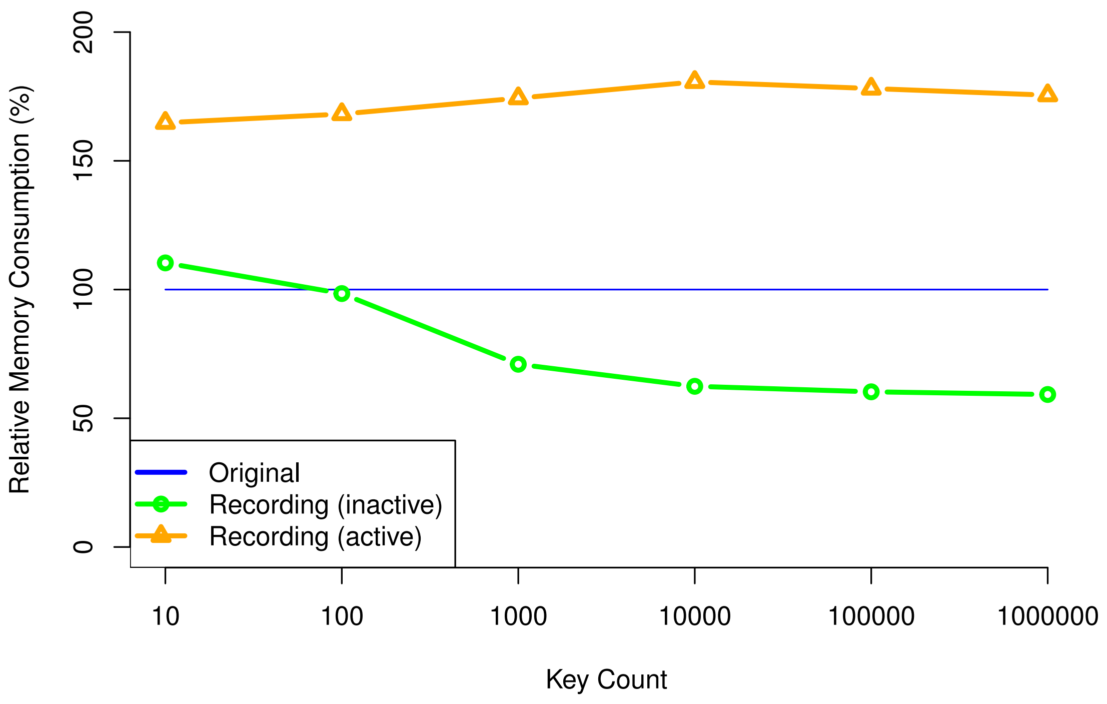


Figure 6.26: Memory consumption relative to the original implementation

When recording is inactive, the memory consumption shows significant enhancements as the number of keys increases, reaching close to a 40% improvement. However, when recording is active, the memory consumption experiences an approximate 80% increase in comparison to the original implementation.

When comparing active recording with inactive recording, there is a significantly larger relative increase, converging to approximately three times for large numbers of keys. We depict this in Figure 6.27.

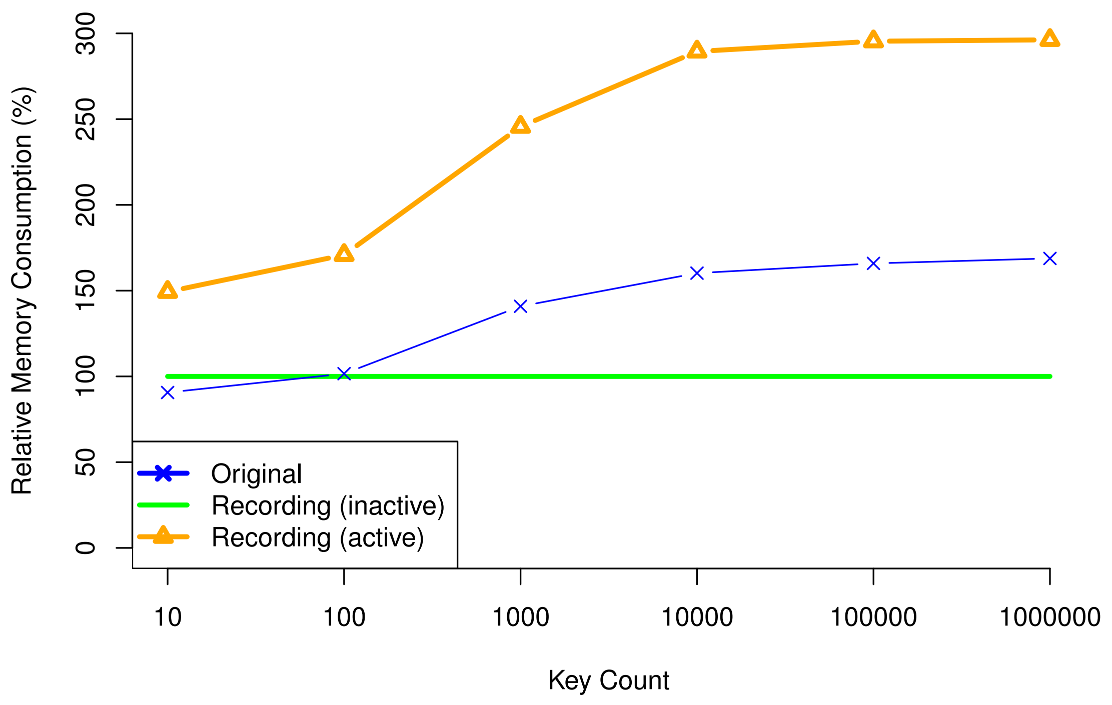


Figure 6.27: Memory consumption relative to inactive recording

When considering the case of 1 000 000 keys, this increase amounts to around 778.31 MiB. The majority of this increase stems from the requirement to assign new names to each key when saving the recording data in Elektra, as described in Section 4.1.2. To differentiate added keys, we prepend the prefix `/elektra/record/session/diff/added`. For modified keys, we need to store both the old and new values, resulting in the prefixes `/elektra/record/session/diff/modified/old` and `/elektra/record/session/diff/modified/new` respectively.

Combining these prefixes with the original key name, such as `user:/test/sw/org/myapp/#0/current/section/subsection/key999999`, leads to an additional 97 bytes of raw data for added keys and 104 bytes (twice) for modified keys. Additionally, in Elektra, every name is stored in both escaped and unescaped forms. Although the specific details of this process are not relevant to our observations, it effectively doubles the memory required for each name, resulting in 194 bytes for each added key and 208 bytes (twice) for modified keys. The data structure responsible for holding the key itself consumes 32 bytes, and the structure for the key name consumes 40 bytes. Thus, for every added key, we have a total of 266 bytes ($32 + 40 + 194$), and for every modified key, we have an extra 560 bytes ($32 + 40 + 208$ multiplied by two).

Modified keys also have modified values, which contribute an additional 20 bytes for the value itself and 24 bytes for the data structure holding the value, totalling 44 bytes.

In the benchmark scenario, we have 1 000 000 added keys and 500 000 modified keys. This results in a total memory consumption of $(1,000,000 * 266) + (500,000 * (560 + 44))$ bytes, equivalent to $568,000,000$ bytes or 541.69 MiB.

Hence, we observe that approximately 70% of the overall increase in memory consumption for the 1 000 000 keys case is attributed to the necessary overhead of storing modified data and persisting the recording information within Elektra. The other 30% may present viable optimization potential.

CHAPTER 7

Evaluation of Requirements Compliance

In this chapter, we assess the extent to which our approach satisfies the requirements outlined in Chapter 3.

FR 1 *Record any changes made to the configuration*

Our case study effectively demonstrated that our approach is capable of capturing all user-initiated changes. We have shown that changes made to applications integrated with Elektra, as well as those unaware of Elektra's existence, can be reliably recorded.

FR 2 *Record a subset of the configuration*

As discussed in Section 4.3, the *kdb record-start* command allows users to specify which sections of the configuration should be recorded, thereby fulfilling this requirement.

FR 3 *Control when recording is active*

The case study and Section 4.3 establish that users have complete control over when session recording is active or inactive. The *kdb record-start* and *kdb record-stop* commands enable starting and stopping recording at will. Furthermore, the Ansible module enables automated control of recording during rollouts.

FR 4 *Apply the changes to other systems*

The case study effectively demonstrated our capability to capture configuration changes on one system, export them, and successfully apply them to other systems, fulfilling this requirement.

FR 5 *Apply a subset of the changes to other systems*

We discuss in Section 4.3 that the configuration export command can be customized to include only specific parts of the configuration, thereby meeting this requirement.

NFR 1 *Transparency to the user*

As we demonstrated in the case study, the user is not required to interact with the session recording feature, apart from starting, stopping and exporting the recorded changes. The powerful *kdb editor* tool even allows using the user's favorite editor to be used to edit the configuration files directly. We clearly showed that, as far as configuration is concerned, the workflow remains uninterrupted, thus fulfilling the requirement.

NFR 2 *Transparency to the applications*

As explained in Chapter 4, the session recording feature is seamlessly integrated into Elektra without requiring modifications to the application's codebase. Even applications not natively integrated with Elektra can benefit from session recording by mounting their configuration files. Hence, this requirement is satisfied.

NFR 3 *Minimal overhead when not in use*

Through our benchmarking, we have demonstrated that session recording does not introduce any noticeable overhead unless actively used. Moreover, we have managed to decrease Elektra's peak memory usage compared to its previous state. Thus, this requirement is successfully fulfilled.

NFR 4 *Concurrent process safety*

Elektra's existing file-locking mechanism, designed to prevent simultaneous configuration modifications by multiple applications, has been extended to meet the demands of session recording. As a result, this requirement is fulfilled.

NFR 5 *Accuracy and completeness*

As outlined in Chapter 4, our approach employs a direct comparison between what the application reads from the configuration database and what it writes into the configuration database, ensuring the reliable detection of all configuration changes. Hence, this requirement is fully satisfied.

CHAPTER 8

Conclusion

Through our case study, benchmarks and evaluation of requirements, we have established that Record Elektra effectively fulfils the defined requirements and offers several key advantages.

The integrated automation using Ansible playbooks enables effortless commissioning of new hosts with the same or similar configurations, as well as updating the configuration of already existing systems. Another identified key advantage is the now built-in undo feature, that allows for safe experimentation and easy rollback to a known working previous configuration state. Elektra's support for validating configuration values ensures correctness and its ability to restart services upon configuration changes is highly beneficial. We have also shown that the solution can be applied to manage software and services that are not natively integrated with Elektra by directly mounting their configuration files using a supported storage plugin. However, we have also identified a limitation related to working with host-specific values when using certain storage plugins.

In terms of runtime performance, we observed that our changes have minimal impact when recording is disabled, and the runtime performance is even better than before for larger configuration sizes. Enabling recording introduces a noticeable performance impact, resulting in approximately 2.5 times longer runtime. Memory consumption also shows improvements when recording is inactive, but a significant increase when recording is active. However, we find that the additional functionality is well worth these trade-offs.

Overall, Record Elektra provides a reliable and efficient solution for configuring and implementing iterative changes in configurations. It offers powerful new features, such as exporting only the changed configuration and automation capabilities, making it a valuable tool for managing configuration changes in various systems.

8.1 Future Work

While Record Elektra has demonstrated its effectiveness in capturing and managing configuration changes, there are several areas of potential future work that can further enhance its functionality and performance.

Firstly, optimization opportunities can be explored to improve the runtime performance and memory consumption when recording is enabled. As we have shown by analyzing the callgraph, disabling unnecessary hooks, such as the *spec* hook, specifically for session recording could noticeably reduce overhead. Additionally, optimizing operations related to building keysets and renaming keys can further improve performance.

Secondly, the reliance on storage plugins for persisting metadata can be a major drawback, as we have demonstrated by our host-specific values in the case study. Finding a solution that allows metadata storage for all storage plugins would eliminate the need for manual editing of exported Ansible playbooks in such cases. We already envision the possibility of a fallback storage mechanism for unsupported metadata in such cases.

Additionally, we have laid the groundwork for expanding the scope of session recording beyond Elektra. We believe it should be feasible to monitor and capture changes to configuration files without relying solely on the *kdb editor* command.

By addressing these areas of future work, Record Elektra can continue to evolve and improve, offering even more robust configuration management capabilities for various use cases.

List of Figures

Bibliography

[1] RedHat, Inc., "Ansible is Simple IT Automation." `https://www.ansible.com/`. Accessed on 2022-02-11.

[2] Progress Software Corporation, "Chef Software DevOps Automation Solutions | Chef." `https://www.chef.io/`. Accessed on 2022-02-11.

[3] Puppet, Inc., "Powerful infrastructure automation and delivery | Puppet." `https://puppet.com/`. Accessed on 2022-02-11.

[4] Docker, Inc., "Empowering App Development for Developers | Docker." `https://www.docker.com/`. Accessed on 2022-02-11.

[5] The Kubernetes Authors, "Kubernetes." `https://kubernetes.io/`. Accessed on 2022-02-11.

[6] Elektra Initiative, "ElektraInitiative." `https://www.libelektra.org/`. Accessed on 2022-02-11.

[7] M. Raab, "Improving system integration using a modular configuration specification language," in *Companion Proceedings of the 15th International Conference on Modularity*, pp. 152–157, 2016.

[8] M. Burgess and A. L. Couch, "Modeling next generation configuration management tools.," in *LISA*, pp. 131–147, 2006.

[9] P. Anderson, "System configuration," Usenix, 2006.

[10] P. Anderson, "Towards a high-level machine configuration system," in *In Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, 1994.

[11] M. Burgess *et al.*, "Cfengine: a site configuration engine," in *in USENIX Computing systems, Vol*, Citeseer, 1995.

[12] Northern.tech AS, "CFEngine." `https://cfengine.com/`. Accessed on 2022-05-09.

[13] M. Raab, "A modular approach to configuration storage," Diploma thesis, Vienna University of Technology, 2010.

[14] M. Raab, B. Denner, S. Hahnenberg, and J. Cito, "Unified configuration setting access in configuration management systems," in *Proceedings of the 28th International Conference on Program Comprehension*, pp. 331–341, 2020.

[15] B. Denner, "Configuration management with libelektra," Diploma thesis, Vienna University of Technology, 2018.

[16] T. Waser, "Configuration management with ansible and libelektra," Bachelors thesis, Vienna University of Technology, 2019.

[17] D. A. Patterson *et al.*, "A simple way to estimate the cost of downtime.," in *LISA*, vol. 2, pp. 185–188, 2002.

[18] D. Hall, *Ansible configuration management.* Packt Publishing, 2013.

[19] J. Hess, "etckeeper." `https://etckeeper.branchable.com/`. Accessed on 2022-04-28.

[20] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 190–200, ACM, 2011.

[21] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, 2017.

[22] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[23] J. Cito, "docker-record - A Semi-automated Approach from Container Setup to Dockerfile." `https://speakerdeck.com/citostyle/docker-record`. Accessed on 2022-04-25.

[24] J. Cito, "docker-record." `https://github.com/citostyle/docker-record`. Accessed on 2022-02-11.

[25] K. Pohl, *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam-foundation level-IREB compliant.* Rocky Nook, Inc., 2016.

[26] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*, vol. 5. Springer Science & Business Media, 2012.

[27] C. Ebert, "Putting requirement management into praxis: dealing with nonfunctional requirements," *Information and Software Technology*, vol. 40, no. 3, pp. 175–185, 1998.

[28] D. Firesmith, “Using quality models to engineer quality requirements.,” *Journal of Object Technology*, vol. 2, pp. 67–75, 09 2003.

[29] The Valgrind Developers, “Massif: a heap profiler.” `https://valgrind.org/docs/manual/ms-manual.html`. Accessed on 2023-04-16.

[30] J. Alberdi-Rodriguez, A. Rubio, M. Oliveira, A. Charalampidou, and D. Folias, “Memory optimization for the octopus scientific code,” 2015.

[31] N. Nethercote, R. Walsh, and J. Fitzhardinge, “Building workload characterization tools with valgrind,” in *2006 IEEE International Symposium on Workload Characterization*, pp. 2–2, IEEE, 2006.

[32] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[33] The Valgrind Developers, “Callgrind: a call-graph generating cache and branch prediction profiler.” `https://valgrind.org/docs/manual/cl-manual.html`. Accessed on 2023-04-16.