Optimizing regression benchmarking for network video products

Thomas Rodenberg

ISSN 1650-2884
LU-CS-EX: 2022-08

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-08

Optimizing regression benchmarking for network video products

Optimering av regressions benchmarking av nätverkskameror

Thomas Rodenberg

Optimizing regression benchmarking for network video products

Thomas Rodenberg
fte13tro@student.lu.se

March 3, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Amilcar Molina, Amilcar.Molina@axis.com
Masoumeh Taromirad, masoumeh.taromirad@cs.lth.se

Examiner: Per Runeson, per.runeson@cs.lth.se

Abstract

Monitoring and maintaining the quality of the software is a critical component of the software development process. Identifying which changes to a codebase introduce performance regressions is essential to address the issues appropriately. However, the process of identifying these changes is increasingly hard with the growing rate of software release and development.

This thesis is a case study investigating a method of identifying the causes of performance regressions in the context of large-scale benchmarking of in-development firmware. Using version control logs and benchmarking tools for the codebase in question, the method seeks to minimize human intervention in the identification process. The result was a tool that performs benchmarks for a set of custom-built firmware. The process assesses at which point in the control logs a change in performance was introduced. Ultimately, the method aims to increase the efficiency and developer-feedback of regression benchmarking in an applied context.

Keywords: performance benchmarking, regression benchmarking, regression cause identification, benchmark automation

Acknowledgements

I would like to thank Masoumeh Taromirad and Per Runeson at LTH. Both of whom have been helpful with all things regarding the writing of the report as well as always providing tips and guidance. I would also like to especially thank my supervisor at Axis Communications, Amilcar Molina, for always helping me and answering my questions. Further, I would also like to express my gratitude to Hampus Sandell, Ola Söder, Madhavi Kagganti, and all the other members of the internal benchmarking team for their support and eagerness to help during this thesis. Additional thanks to Tomas Westling along with the rest of the people at Axis, for their help, friendly reception and inspiration.

Contents

Chapter 1
Introduction

1.1 Problem formulation

In the software development process, monitoring and maintaining the quality of the software is critical. However, maintaining a high level of quality is increasingly hard with the growing rate of software release and development. Software systems failures are often due to performance issues rather than functional bugs, of which performance regressions are often the most critical. [9] The introduction of performance regressions is a common occurrence that needs to be dealt with in a timely manner to keep up the momentum of the development. Typical performance regressions include increased CPU or memory resource utilization, increased system boot time, and degradation of response time.

Identifying the changes that introduce regressions to a codebase is critical to continuously assure a certain quality standard of the software in development. This is because it is not until the cause is identified that a regression can be appropriately dealt with. It often remains a task carried out after a system is built and deployed in the field or dedicated performance testing environments. Large amounts of resources are required to detect, locate and fix performance regressions at such a late stage in the development cycle. Finding ways to identify the cause of performance changes while minimizing manual human intervention is a field of study that is of high interest. The number of required resources would be significantly reduced if developers were notified whether a code change introduces performance regressions earlier during development.

1.2 Goal

This project aims to investigate and provide a suitable way for identifying the code-commit that is responsible for a given regression. Addressing this problem will make the development of new firmware easier for the development teams since they have an entry point to the cause of the performance problem. Ultimately the product of this work should be a deployable tool that potentially could be run daily together with Axis current benchmarking suites or serve as a *Proof of Concept* for a future potential tool. Additionally, this report will present a case study in how to automate regression cause identification while answering and discussing the following questions:

RQ1: How is the benchmarking carried out currently at Axis? What are their main concerns?

RQ2: What would be the main functionalities of the target solution, and how would it address the relevant challenges?

RQ3: In what ways would the solution affect their current practices? Is it a feasible tool for Axis to utilize, and if so, in what capacity?

1.3 Case company

Axis Communications AB is a leading producer of network-oriented technology with a vast array of products, mainly consisting of surveillance cameras. The company was founded in Lund 1984, which still houses its principal office and headquarters. As of 2021, they employ over 3800 people in more than 50 countries worldwide. Since they outsource their manufacturing, the main focus of their business is the design and development of the hardware and software of their products. This makes software quality key to their success, and a lot of resources are directed into quality assessment of various kinds.

A large and growing base of video network devices means new products are constantly developed and software updates are regularly released. To validate the devices' quality and functionality, extensive testing is performed continuously. New firmware is built daily and is then benchmarked to detect changes in the performance as soon as possible. Since they have a large base of devices receiving software updates regularly, it is very hard to effectively identify and diagnose the root cause of performance regressions, which is currently done manually. A promising improvement is to automate the process of identifying the cause of regressions, and hence, Axis looks for ways to implement this within their benchmarking practices.

Chapter 2
Methodology

2.1 Research method

Van Aken [28] claims that a research paradigm refers to the combination of the types of research questions, the methodologies allowed to answer them, and the nature of the pursued products. He further claims that scientific research can be separated into three major paradigms:

1. Formal sciences
2. Explanatory sciences
3. Design sciences

Within this framework, formal sciences refer to empirically void fields such as mathematics, while explanatory sciences describe the empirical sciences. However, the goal of design science is to develop knowledge for the design and development of artifacts. Examples of the classes of problems design science aim to address are; construction problems or improving the performance of existing entities, that is, to solve improvement problems. For example, engineering sciences and medical science fall into the realm of design science.

Since software and software development are designed artifacts, Runeson et al. [22] argue that the research related to software (and software development) can be appropriately framed as design scientific. However, the boundary between explanatory and design sciences is not always clear. Research often includes elements explaining a naturally occurring phenomenon for which an intervention is designed and validated. However, the software engineering research ultimately aims to develop and validate practically practical methods, technologies, and tools to provide improved software engineering practices for industry.

The design scientific approach can divided into the following sub components:

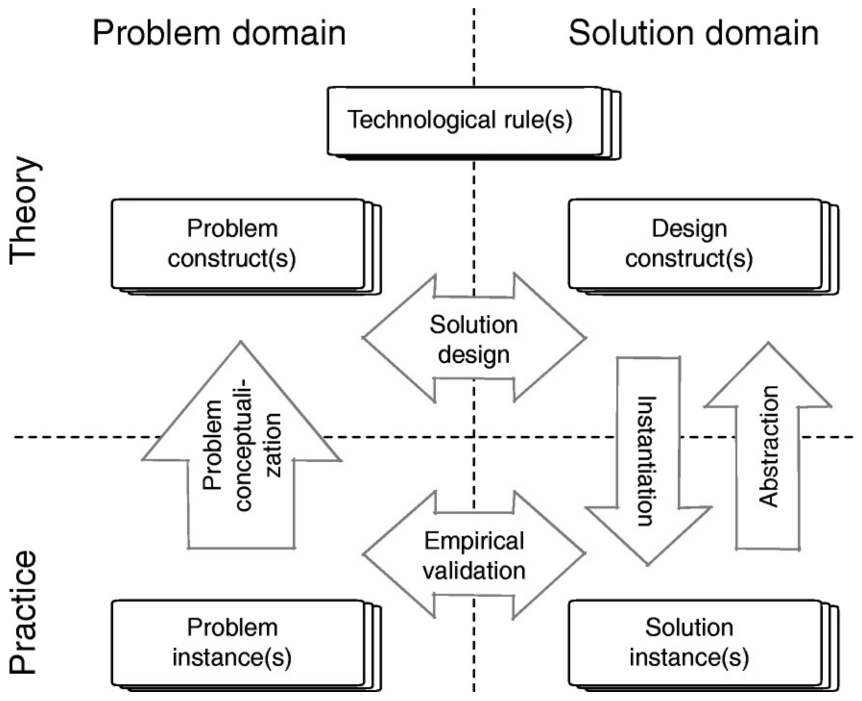

Figure 2.1: An illustration of the interplay between problem and solution as well as between theory and practice in design science research." by Engström et al. "How software engineering research aligns with design science : a review." [13], licensed under CC BY 4.0

Problem conceptualization is the activity of analyzing the problem, identifying its constituent parts and the context of which it is part. While it serves as a basis for the research activity, it is not a purely descriptive endeavor and is often intertwined with the activity of solution design.

Solution design refers to the activity of formulating a solution to the problem at hand. Here, many alternative solutions, previous research, and case studies are considered.

Instantiation refers to the activity of implementing a solution to a specific problem.

Abstraction refers to the activity of describing the key design decisions for a solution.

Empirical validation refers to the activity of evaluating the implemented solution in its context. The primary goal of empirical validation is to assess whether the proposed solution is feasible for the given problem. The scope of the design knowledge gained in a study can be extended by systematically extending the validation scope in subsequent studies.

This model is, however, not strictly ordered, and different activities can be revisited multiple times during the course of research. Design science is a paradigm used in many different research fields. It is instantiated in many different variants. These points reflect mainly what is most relevant within software engineering. Figure 2.1 depict how the design scientific activities can interact in order to create knowledge.

Each of these activities was applied to some extent within the execution of the thesis. However, the activities that were of main focus were problem conceptualization and solution instantiation. Solution design and instantiation were conducted in unison while, at each step, empirically validating each iteration. Once a proper instantiation was made, the activity of abstraction and a more systematic validation were conducted.

2.2 Research components

2.2.1 Literature review

An essential step of planning and conducting a research project is first to review the available literature on the intended subject. To do this, Thiel [27] proposes a series of important steps:

1. Keyword searching
2. Selection of relevant papers
3. Review of paper abstract for relevance
4. Review of complete paper for relevance
5. Critical analysis of the results as they apply to the new research project

In this project, these steps were followed when selecting relevant papers. This was done with keywords such as; *performance regression, benchmarking software, regression benchmarking, performance testing, software performance, microbenchmarks software, regression cause identification.* When read, the papers found were organized according to relevance. When the most relevant papers were identified their references were reviewed for further information.

2.2.2 Interviews

In case studies, interviews are often an important source of data. In interview-based data collection, a series of questions are asked to a set of participants about the areas of interest in the case study. These interviews can be divided into unstructured, semi-structured, and fully structured. [21] In an unstructured interview, the interview conversation will develop based on the interest of the subject and the researcher. In a fully structured interview, all questions are planned in advance and asked in the same order as in the plan.

In a semi-structured interview, questions are planned but are not necessarily asked in the same order as they are listed. The development of the conversation in the interview can decide in which order the different questions are handled and allow for certain improvisation and exploration of the studied subject. Further, concerning the scope of the questions, interviews can be structured according to three general principles: the funnel, the pyramid and the time-glass. [23] The funnel model begins with open questions and progresses towards more specific ones during the course of the interview. The pyramid model begins with specific questions but moves towards open questions. The time-glass model begins with open questions, tightens the structure in the middle, and opens up again at the end of the interview.

The main goal of conducting interviews at Axis was to better understand how their benchmarking is conducted, how performance issues are identified and dealt with, and how they hope to improve that. Three semi-structured interviews were conducted with two members of the internal benchmarking team and the platform manager, who all deal with these issues daily. All interviews in this thesis were held in person and were not recorded. The structure of the interviews was based on the time-glass model. **Appendix A** presents the general interview outline, including the questions that all the interviews had in common. Each interview focused on different aspects of the problem and included different questions with different overall scopes.

2.2.3 Problem conceptualization

Each problem needs to be understood within its conceptual framework. The outcome of the problem conceptualization is expressed in terms of a set of problems with a corresponding set of target solutions. For instance, a problem could be described in terms of a group of target users, their questions and tasks, and their measurements or data. Therefore, problem conceptualization is closely connected to the solution design and can usually not be performed in isolation. Further, the conceptualization often needs to be repeated at different abstraction levels depending on the solution. The first level would be the stakeholder's problem description and, in the case of a tool, reaching the level of implementation details. [22]

In order to successfully carry out this project, a sufficiently broad conceptualization of the current methods of regression cause identification in use at Axis was needed. A set of criteria for a target solution could eventually be formulated from the literature review, the interviews and a review of available written material internally at Axis. Insights into the problem were also gained from attending daily meetings held with the internal benchmarking team, where the latest benchmarking results were discussed.

2.2.4 Solution

By its nature, the theoretical contributions of design science research are context-dependent. The knowledge gained consists of prescriptive recommendations most often captured in the concept of technological rules. [22] Van Aken defines a technological rule as "a chunk of general knowledge, linking an intervention or artifact with a desired outcome or performance in a certain field of application." This means that the prescribed intervention is not exclusively for a specific situation but a general prescription for a class of problems, although it might be a limited one. [28]

The design knowledge within the technological rule aims to help software engineering professionals design customized solutions to their specific problems. One could generalize it to be the stakeholder's desired effect of applying a potential intervention in a specified context. Expressing this explicitly can help identify and communicate the core value-creating aspects of the research. [22]

To be able to provide an answer to **RQ2** and **RQ3** a target solution to the problem was needed. An instantiation of this target solution was made during the project, where the implementation was made in phases. Different parts of the final solution were produced and assessed separately at first. Much of the implementation was heavily based on the different tools researched within the scope of the problem conceptualization. This meant that the activities of solution design and implementation were partly performed in conjunction with each other.

2.2.5 Evaluation

Within the design scientific paradigm, the intervention is the object of the validation study, exposing the product of the project to its intended context. The context refers to where the research is conducted, and the expected effect defines the validation criteria. This indicates that a real software engineering context as a validation context is suitable for design science research. For this reason, case studies have been brought forward as the natural research methodology in design science. Extending the scope of the validity for intervention is done by either applying it to new contexts or by reasoning about its validity within another context by comparing its key characteristics. [22]

One of the results of the conceptualization was a specification of requirements for this thesis project. Evaluating whether these were met, however was not an inherently obvious process. An empirical validation of the solution in its intended context was made utilizing the backlog of known performance regressions. This approach of evaluating the solution was informed by the results of the interviews conducted at Axis. The results of this were to be used to validate whether the intended functionalities of the target solution, with respect to **RQ2**, were present and, additionally, serve as basis for a discussion regarding **RQ3**.

Data was gathered with the goal of evaluating the overall utility of the solution. The produced data consisted of:

- The results from the benchmarks conducted during execution of the solution.
- The resulting conclusions reached (The identified cause, or lack there of).
- Additional data relevant for the process and its execution (which commits were built, benchmarked and in which order).
- The time spent executing the distinct stages of the process.

Assessing the validity of the results could not be done objectively since known regression causes were not available for the corresponding backlog of regressions. Evaluation of the individual results were therefore to a large extent done manually.

Chapter 3
Background and related work

3.1 Performance and performance regressions

In the ISO 25010 standard, performance efficiency is defined as the performance relative to the amount of resources used under stated conditions. This characteristic is one of eight key software quality components and is composed of the following sub-characteristics: [7]

Time behaviour - degree to which the response and processing times and throughput rates of a product or system meet requirements.

Resource utilization - degree to which the amounts and types of resources used by a product or system meet requirements.

Capacity - degree to which the maximum limits of a product or system parameter meet requirements.

Although performance regressions are not all bugs, they usually have a direct impact on the experience of a system and large software systems failures have been found to more often be due to performance issues rather than functional bugs. Dealing with performance regressions remains a task that is carried out after a system is built and deployed in the field or dedicated performance testing environments. This means that large amounts of resources are required to locate and fix performance regressions at such a late stage in the development cycle. The amount of required resources can be significantly reduced if developers are notified whether a code change introduces performance regressions earlier during development. [9]

3.2 Related work

Within the context of industry, the goal of regression testing is generally to get confidence that the system changes have not introduced unwanted system behavior rather than to find explicit errors. The regression testing scope depends on the projects' timing and risk analysis for incoming changes. Minhas et al. concluded that test automation is mainly reliant on in-house tools, allowing for faster and better alignment between the testing process, testing strategy, and tool support. [19] Further, test strategy definitions were often to be an ad hoc practice among the companies involved in the study, which confirms the need for in-house and flexible tooling.

Efforts to, with greater precision, identify and detect regressions in software performance have been many and ranged from a wide array of different approaches. In a study by Chen and Shang, [10] a statistical performance evaluation on a large number of individual commits from ten releases of Hadoop, and additional commits from five releases of RxJava was conducted. After first filtering out the irrelevant commits, they performed benchmarks on four physical performance metrics; CPU usage, Memory usage, I/O read, and I/O write to measure performance. In order to minimize noise, they repeated the execution of their benchmarks 30 times independently and used a statistical approach to evaluate whether or not a performance regression had been introduced. By comparing performance metrics that are measured during the tests or performance microbenchmarks, they were able to identify whether any of the studied commits introduced performance regressions. These efforts were made to collect a sample of performance regression introducing commits for further investigation. [10]

Daly et al. [11] in their used a particular change point detection algorithm called E-Divisive mean to detect performance changes. The implementation was made to a system that runs thousands of benchmarks periodically, usually every 2 hours or 24 hours, producing one or more scalar values as a result. Additionally, they also had to integrate it into an existing performance testing system and visualize the results so that engineers could triage and use the gained information. Through these efforts, they were able to drop their false positive rate for performance changes while qualitatively making the entire process smoother and more productive, catching smaller regressions. [11]

Another study by Shang et al. [25] proposed an approach to automatically detect performance regressions using regression models on clustered performance counters. The approach analyzes all collected counters instead of focusing on a limited number. Some of these performance counters were, for instance, CPU and memory. First, the counters were clustered to determine the number of which was sufficient to truly represent the performance of the given system. Statistical tests were then conducted to select the target performance counters for which the regression models were then built. The models were then applied to new versions of the system to detect performance regressions. The approach could successfully detect both injected and real performance regressions in the two case studies, one open-source and one enterprise. [25]

A study by Heger et al. [15] uses an approach that builds a hybrid regression detection strategy where the main components are bisection of revision graphs and analysis of performance annotated call trees. Their method integrates root cause analysis into the existing development infrastructure using performance-aware unit tests and the revision history. As a part of the approach, the Git bisection algorithm was extended to identify the changes that introduced a performance problem. [15]

Additionally, Alcocer et al. [24] proposed an interactive visualization to compare performance variations caused in a set of software versions and performed controlled experiments to show the viability of their approach when identifying and understanding performance regressions and improvements. [24] Another approach suggests prioritizing test cases in performance regression testing for collection-intensive software, where test prioritization is based on performance impact analysis that estimates the performance impact of a given code revision on a given test execution. [20]

3.3 Benchmarking

In the book *Systems Benchmarking*, Kounev et al. define a benchmark as follows: "A benchmark is a tool coupled with a methodology for the evaluation and comparison of systems or components with respect to specific characteristics, such as performance, reliability, or security." [17] Using the definition of performance expressed in section 3.1, better performance would mean more work accomplished in either shorter time, using fewer resources or with increased capacity. Depending on the context, high performance may involve; high responsiveness when using the system, high processing rate, low amount of resources used, or high availability of the system's services.

Benchmarks are mainly characterized by: workloads, metrics, and measurement methodology. Workloads determine under which usage scenarios and conditions measurements should be performed. The metrics determine what values should be derived based on measurements to produce the benchmark results. Lastly, the measurement methodology defines the overarching process to execute the benchmark, collect measurements, and produce the final results. If one were to limit the tests to some specific system component, that would constitute a microbenchmark. [17]

For best usability, a complete regression benchmarking process should be as automatic as possible so that human attention only is needed when an anomaly has been detected. The requirement of automation means that a machine has to obtain, compile, and deploy both the software under development and the benchmarks, execute the benchmarks on the software under development, monitor the execution, and store and analyze the results. To meet these challenges, Kalibera et al. [16] proposes the use of generic benchmarking environments that supports automated deployment, execution, and monitoring of benchmarks and related software. This should also be applied to repositories for storing data in a standard format that serves as a data source for analysis and visualization tools.

This thesis has been heavily focused performing benchmarks and using their results to draw various conclusions regarding the codebase under investigation.

3.4 Software configuration management

As modern software development often is conducted by large teams and complex organizations, it is common for many versions of the same software to be deployed in different locations or systems. Since its often necessary to develop different parts of the software in parallel, to locate and fix bugs, it is of absolute importance to get, build, and run various versions of the software to determine in which version or versions the problem occurs. [12]

Software configuration management, SCM, is the process of tracking and controlling software changes. A configuration management system must include a history of changes made to each file. When working correctly, this can guarantee the reproducibility of a piece of software across many hosts. Basic SCM requirements encompass the four categories described below.

1. Version control - the process of keeping track of all changes to every file, supporting parallel development by enabling easy branching and merging. Every type of object that evolves in the software development environment must be version-controlled.

2. Environment management - providing a consistent, flexible, inexpensive, and reproducible environment to compile, edit, and debug software. Environment management is the process that selects and presents the appropriate version of each file in a way that enables the development tools to work smoothly.

3. Build management - the process of building software components and also documents the contents of each software build. The documentation must be complete enough and reliable enough to recreate the environment that created the build in the future (for making patches and debugging problems).

4. Process control - a set of policies and enforcement mechanisms to ensure that the software is developed according to a defined development methodology. Process controls include; monitoring, notification, access control, and reporting. These controls must be flexible, enabling development organizations to customize their environment to support their chosen policies. [18]

The concept behind the solution presented in this thesis relies on the bisection of commits in version control logs to draw various conclusions regarding the codebase under investigation. Furthermore, environment management was critical to the implementation of the solution.

Chapter 4
Benchmarking at Axis

This chapter presents an overarching picture of the current benchmarking practices at Axis, the case company. The information presented is a combination of the results of semi-structured interviews, discussions with the projects supervisor and research into the written documentation which has been available.

4.1 An overview

At Axis, much effort and resources are spent ensuring the quality of their products. Different teams ensure different aspects of performance at different levels of development and departments. The internal benchmarking monitors' the performance of a large selection of their products once they are in production. Since firmware for these products is continuously being developed, a system is in place at Axis to build and test new firmware daily. This means that changes can be introduced from any given day to the next.

It is then the *Release Coordinators*' job to look over the results of these benchmarks to assess whether a regression has occurred in any of the relevant metrics. The results are organized in plots similar to figure 4.1, which monitor a set of the most critical metrics. If an issue is detected, it is of high priority to, as accurately as possible, identify the cause and notify the person or team responsible. That way, avoidable regressions can be reverted swiftly and efficiently. Regressions can, however, be caused by factors other than firmware, such as issues with the network, or changes to the benchmarking software. These factors need to be excluded before an issue can be appropriately tracked. New findings are, for this reason, usually discussed in the daily meeting with the benchmarking team before further investigation.

Once a regression is identified, an email is sent to the development team responsible, and if they acknowledge the issue, a ticket (task) can be produced to address it. Issues are tracked through Jira [1], a proprietary issue tracking product developed by Atlassian that allows bug tracking and agile project management. Members of the internal benchmarking team estimate that a weekly average of three potential regressions are detected, of which

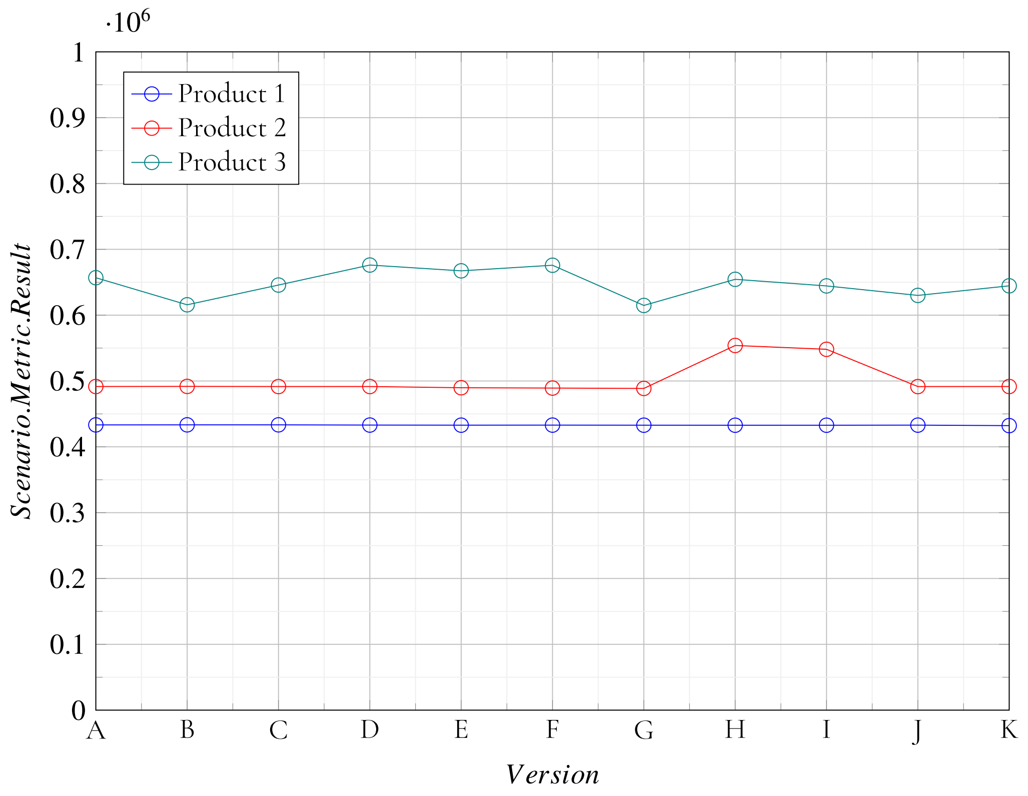

Figure 4.1: An example of how the performance for a given metric can vary for three different products. Product 1 is stable, Product 2 has a change in performance, which is introduced in version H and is reverted in version J, and Product 3 has a far less obvious and noisy performance result.

one usually results in a ticket. Cases where a ticket is not produced are, for instance, when regressions in performance are anticipated, such as when new features are introduced.

Regression findings are noted and kept in a table regardless of whether the issue is addressed, or a ticket is made concerning it. An entry in the table specifies which product or products, firmware version, and metrics a given regression has been observed. The table essentially contains the documented history of notable regressions from the internal benchmarking at Axis. An overview of the general process can be found depicted in figure 4.2.

4.2 The firmware

Firmware is a specific class of computer software that is built into an embedded systems product and stored in nonvolatile memory, such as ROM, EPROM, or FLASH. Firmware is also known in the industry as *embedded software* or *low-level software* since it provides low-level control for a device's specific hardware. Major firmware components optionally include an operating system, kernel, device drivers, and application code. [26]

The firmware referred to in the context of this project is the firmware which is in development at Axis. The firmware are updated five times a week, for each product, through a shared software platform. The software platform controls, builds and maintains the firmware for a all the products that use *Axis OS*. Axis OS is the operating system used by a large selection of Axis products. As described in figure 4.2, each update to any of the products on the platform is made through the platform and each new firmware version is built through the platform.

Every daily firmware update has a version, which is shared by all products and represents a snapshot of the state of the platform at the time the firmware was built. This is practical since large amounts of source code are shared amongst a large number of products. Further, an earlier version of any firmware can easily be accessed and reproduced through the version control history of the platform. Each daily firmware update is the main subject of the internal benchmarks.

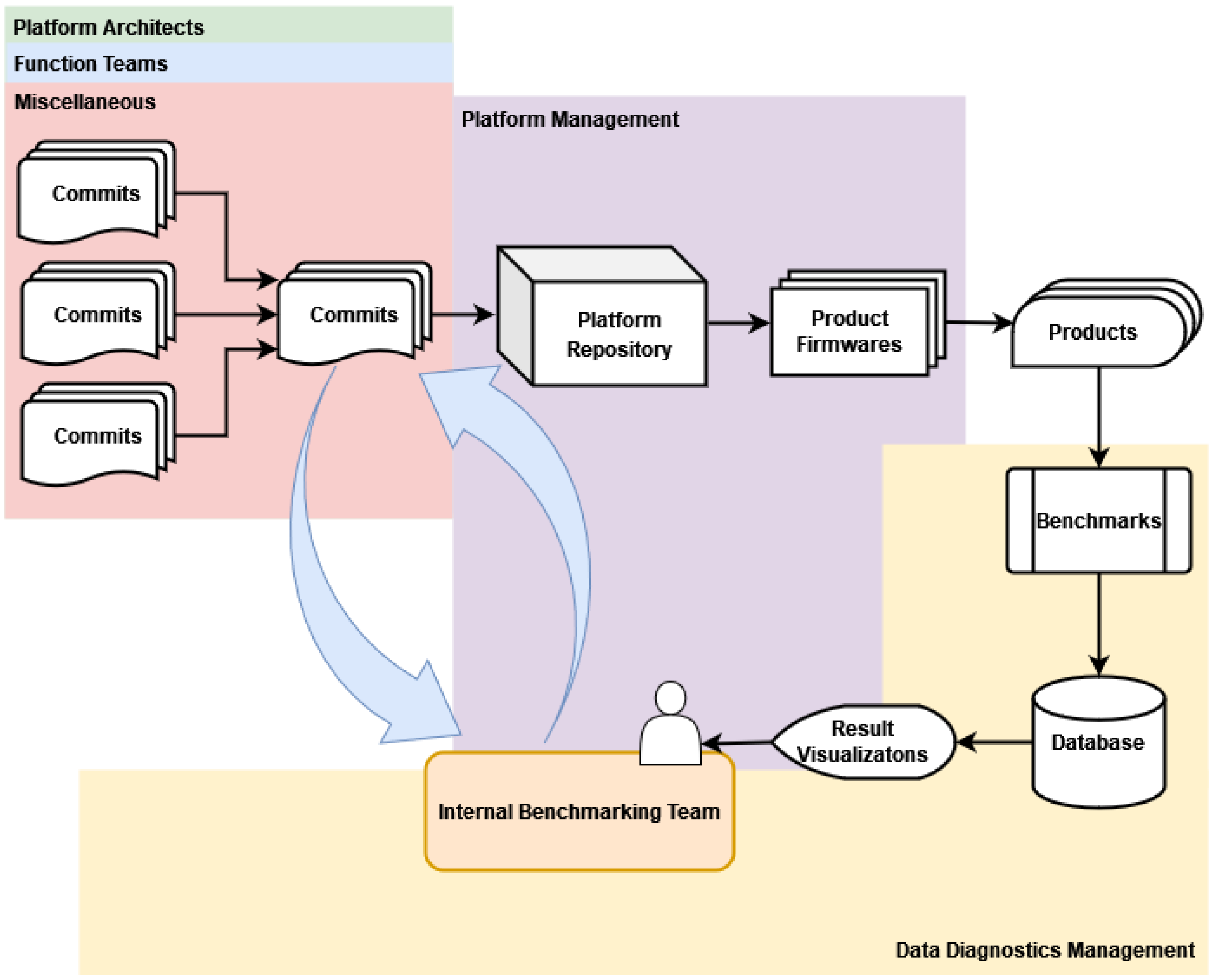

Figure 4.2: The role of the internal benchmarks at Axis is to provide feedback regarding the performance of their products to the developers of their firmware. The coloured backgrounds represent the domain of the department which labels them.

4.3 The stakeholders

At Axis, the department responsible for the internal benchmarks is the *Data Diagnostics Management* department (DDM). DDM's role at large is to gather and analyze data related to Axis products' performance. The internal benchmarking team consists of a group of six engineers from the Data Diagnostics Management department and one engineer from the *Platform Management* department, the *Release Coordinator*. This team conducts daily meetings where the results of the latest benchmarks are reviewed and discussed. The benchmarking team develops and maintains the benchmarking software and makes sure that it runs smoothly every day. The Platform Management department is responsible for managing the *Axis OS*, which partially means making sure that there is a daily firmware build for each product that is working and is stable.

All parties involved in the development of the products at Axis have a stake in how the performance of the products they are working on evolves on a regular basis. As one of the team members puts it when asked whom the stakeholders were for the internal benchmarking: "Everyone." The main stakeholders are, however, the *Platform Architects*, the *Development Teams*, and ultimately, the *Global Product Manager* of Axis OS.

The main channel of communication for the benchmarking team with its stakeholders is maintained by a quarterly report produced by DDM at large, containing information on their activities, including the benchmarks. This ensures that all interested parties are kept aware of their activities and have access to their most important new developments. A meeting is also held in conjunction with this report where important stakeholders and development teams can attend after reading the report. The report is produced by an *Experienced Engineer* from the benchmarking team, and their most significant findings, changes, and issues are included in the report. Additional communication with the development teams is maintained through the Platform Coordinator when performance changes are anticipated or detected.

4.4 The benchmarks

The benchmarking tool currently used at Axis was put into use in 2016 to monitor the performance of their products more efficiently and to be able to catch regressions as soon as they are introduced. Only a handful of roughly 7-20 products with the most functionality were tested on introduction. At the time of writing, however, over 100 different products are being benchmarked and monitored daily. These benchmarks are carried out three times a night, five times a week for all the relevant products. The time consumption of one benchmarking suite is, on average, about 1 hour and 30 minutes. The tests are carried out over the network using an open-source automation tool called Jenkins, which enables scheduling jobs having jobs trigger each other, among other things. [4]

The benchmarking environment is kept as generic as possible, using a standardized codebase for every product, much like what Kalibera et al. recommend. [16] That is, a Docker image is built of the environment and stored in a database, which can then be easily deployed on a Jenkins server. The benchmarks are conducted over the network, which is how the products are utilized in practice. The process iterates through a set of predefined scenarios gathering measurements relating to the performance of the products. Some of the main principles regarding Docker and Jenkins are included in **Appendix B**.

4.4.1 The scenarios

Several different test scenarios are run while extracting the measurements to gain a sufficiently detailed understanding of the realistic performance of the products. These scenarios include, for instance, *idle*, *high-load*, etc. An entire suite usually runs for about an hour and a half, logging over 800 separate metrics over 16 scenarios. For some noisy metrics, measurements are made several times to produce an average result. Each benchmarking scenario focus on a different aspect of the performance of the products. Examples of some of the main scenarios utilized in this project are:

Boot-time: Collects various measurements relating to the startup time of the unit.

Idle: Scenario collecting all measurements while the unit is turned on but idle, with no outside interaction or streams running.

Edge: Scenario measuring resource consumption while enabling motion detection, mainly collecting measurements relating to CPU usage and availability of memory.

High-load: Unit running a high load, collecting measurements relating to CPU usage and availability of memory.

Rtsp-response: Scenario measuring response time and latency with regards to the real-time streaming protocol.

The benchmark scenarios can also be executed in isolation, which means that if one is interested in validating the performance of a specific metric, it would not be necessary to run the entire one and a half hours of benchmarks to do so. These are what will be referred to as *microbenchmarks* in this report.

4.4.2 The results

The results produced by the benchmarks are stored in JSON files, which are loaded to a database, and inspected through a visualization tool. The visualization tool is highly customizable, making it easy to adjust and filter what is monitored, which is important when tracking the results of over 100 products. The primary metrics being closely monitored are; *boot-time, system ready time, CPU-usage, flash utilization, real-time streaming latency, memory available, firmware size, and parameter response time.*

For easier inspection, some metrics are tracked as groups based on which integrated circuits the products operate. Since these products share most of their hardware this usually means that they share large amounts of their functionality and software as well. Tracking groups of products can be helpful when the results of a metric are particularly noisy since the results can get an average of a more significant number of measurements.

4.5 Project specification

From the various interviews and discussions with the projects supervisor at Axis, it became clear that the focus of this thesis would be on aiding the identification of the cause of regressions in the benchmarked products' performance. The units of possible causes of regressions, in this case, were the individual commits to the Axis OS platform. Identifying the cause of a regression would incorporate; building firmware from commits to the platform and benchmarking these to be able to pinpoint which commit introduced the regression.

The process was to be fully automated while being available as a tool at the disposal of the people for whom it would be relevant to use. The tool should be containerized in such a way that it is accessible and executable on an automation server, similar to their existing benchmarking practices. That way, anyone can utilize it whenever a noticeable regression has occurred with minimal effort. The main criteria for the thesis implementation eventually boiled down to the following:

1. Benchmarks conducted on firmware built from intermediate commits to the Axis OS platform

2. Automated search for the commit where a given performance regression was introduced.

3. An implementation containerized and deployed on an automation server.

These criteria relate to the second question raised by **RQ1**, the main concerns with the benchmarks and how to develop their process further. The criteria also serve as a basis for an answer to the first question in **RQ2**, addressing some of the main functionalities of the target solution.

Chapter 5
The solution

This chapter presents an overview of the solution to the problem this thesis sets out to address. To aid in the abstraction of the overall solution, it has been separated into a set of distinct components.

5.1 The design

The result of this project is a tool that automatically performs a given benchmark for a set of custom-built firmware to pinpoint where a change in performance has been introduced. Much like Chen and Shang's study [10], the design utilizes the concept of running microbenchmarks on commit level to be able to evaluate where regressions in performance have been introduced. Since the benchmarking tool in current use at Axis has the option to run each scenario individually, these were utilized to reduce the tool's overall resource consumption.

Similar to the study by Heger et al., [15] the tool uses a variation of the git bisection concept to find which commit in the version control logs to evaluate next. Firmware from the commits intermediate to the two relevant versions is built. These are then benchmarked to assess which direction in the version control logs to branch next. The search process is repeated until two adjacent commits have been benchmarked, and the search is unable to branch further. At that point, the solution will have an answer to where the regression is introduced in the version control logs.

To further meet the **specification** the tool should be simple to use and executable on an automation server. This meant containerizing it so that it could be run from virtually any environment. Containerizing the source code together with a suitable environment is helpful for the maintainability of the project and its ease of use. The structure of the process is a straightforward procedure that takes a set of input parameters based on which it goes through a set of predefined stages necessary to reach a conclusion. The procedure has divided into two parts: Setup and Search.

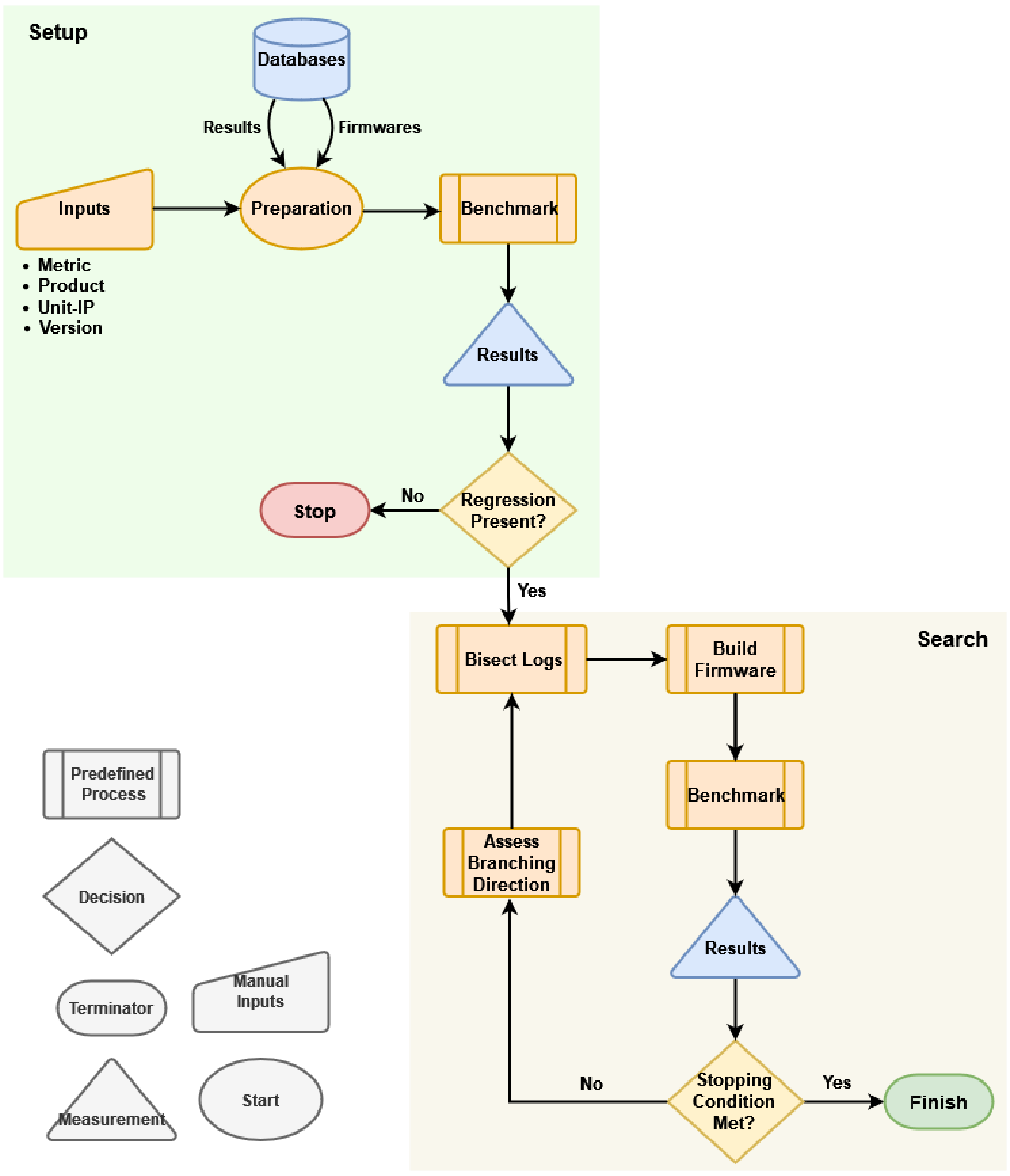

Figure 5.1: The execution flow overview of the solution.

5.2 Setup

The setup stage handles input information, loads needed information from the databases, and verifies whether or not the performance regression from the previous benchmarking results can be reproduced. When triggered, the setup also performs a cleanup and preparation of the tools environment. A concise description of the setup process is depicted in the upper half of figure 5.1. The main components of the setup are presented in the following subsections.

5.2.1 Input

For a tool of this nature to be able to run properly, certain information has to be provided. The input parameters needed for the tool to execute are:

Metric - specifying for which benchmarked metric the regression has been detected.

Product - specifying which product model to use.

Unit IP - specifying the IP-address for the unit to use.

Version - specifying for which version of daily development firmware the regression first was observed.

The product and metric are needed as input parameters to get the reference results from the database, which contains the benchmarking results. Since the benchmarks are conducted over the network with a physical machine, the IP-address of a specific unit is needed. However, the IP-address parameter is provided automatically when the tool is executed on the automation server.

5.2.2 Database interaction

All results from the internal benchmarks are stored in a database. The old results for the relevant versions and product are needed as a reference to evaluate any new benchmarking results. These results are retrieved from the database. Firmware for the two relevant versions can are also retrieved from a separate database which saves a significant amount of time compared to the process of building them.

5.2.3 Reproducing the regression

To properly search for the cause of regression, benchmarks for the previously run firmware versions need to be executed again to see if the regression can be reproduced. The main reason for this is that regressions can be introduced by factors other than the changes in the firmware, such as broken product units, bad network connection, or changes to the benchmarking software. To better filter out irrelevant causes of regressions, the microbenchmark associated with the given metric is executed. These results are then compared to the ones previously gathered in the regular benchmarking suites.

5.2.4 Evaluating results

When the tool is triggered, it is assumed that a regression is present in the reference results for the given product and metric. The results of each new microbenchmark are compared to reference results. The reference results correspond to an average of the results from the nightly benchmarks for the given version and metric. A regression is considered reproduced if the results show that:

$$|A_{ref} - A_{tool}| < |B_{ref} - A_{tool}| \land |B_{ref} - B_{tool}| < |A_{ref} - B_{tool}|$$

or

$$|B_{tool} - A_{tool}| > |B_{ref} - A_{ref}| \times 0.8$$

Where B represents the current version that demonstrated a regression and, thus, is under the investigation. A represents the version preceding it. A_{ref} denotes the old result (from the database) for version A and B_{ref} for version B. A_{tool} and B_{tool} denote the new results from the microbenchmarks conducted as a part of the tool's execution. All results correspond to the benchmarking metric specified in the input.

Since the assumption is made that a regression is present in the reference results, the overall evaluation is kept simple. The evaluation also supports missing results since a firmware update might have caused the error.

5.3 Search

If the same regression is reproduced, the goal is to search for the commit responsible for the regression. The method used to pinpoint where the regression was introduced is a variation on the binary search algorithm. The process aims to, in an iterative manner, branch closer and closer to the commit which introduced the regression. For each step of the search, a new firmware has to be built, and a new microbenchmark has to be conducted. A concise description of the search process is depicted in the lower half of figure 5.1. A visualization of the search process is presented in figure 5.2 The main components of the search are presented in the following subsections.

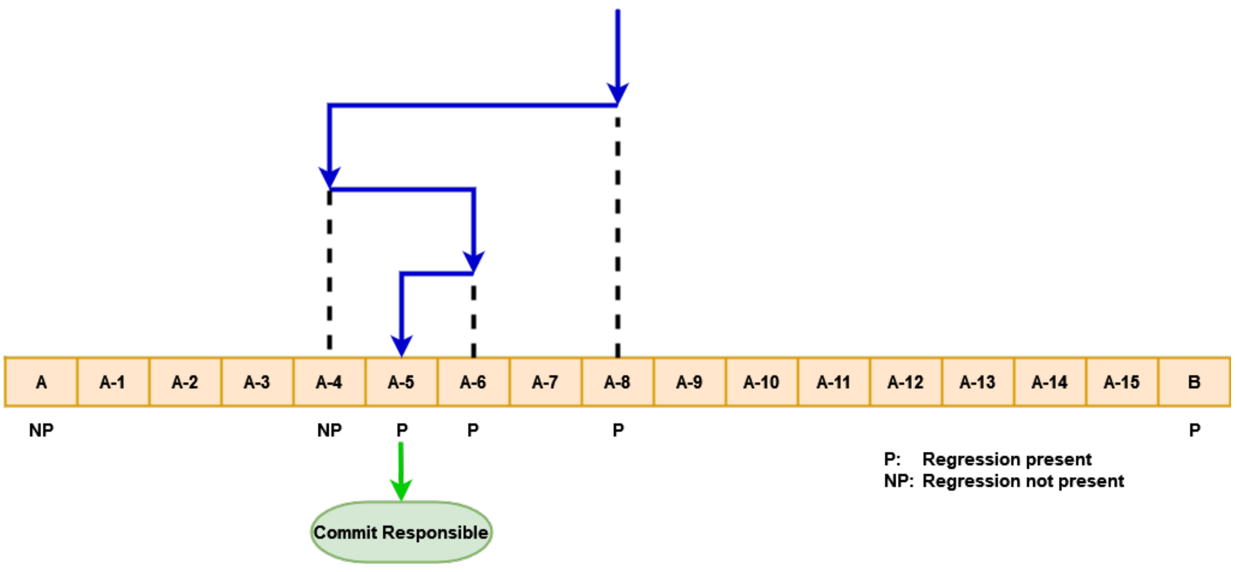

Figure 5.2: The search process for the regression cause identification process. A and B are firmware versions and A-1 to A-15 are the individual commits which separate them.

5.3.1 Bisection

Firstly, the platform repository is cloned, and the commit logs for the relevant versions are extracted and processed. As the title suggests, a commit at the center of two previously benchmarked points in the version control history is selected. Which two points to bisect depends on which "direction" the latest result indicate according to the branching condition. This principle is shown in figure 5.2. The commits to the platform are all the same regardless of which product or metric, is dealt with and within the scope of this project, all commits are treated as equally relevant. The concept is based on the *Git Bisect* functionality in git, for which a brief summary is included in **Appendix B**.

5.3.2 Branching conditions

With each consecutive benchmark result, the search continues, branching to either before or after the latest benchmarked commit based on whether the regression is present or not. Similar to the earlier evaluation, a regression is considered reproduced if the results show that:

$$|A_{tool} - C_n| > |B_{tool} - C_n|$$

Where A_{tool} and B_{tool} once again denote the results from the microbenchmarks for the specified metric, conducted as part of the tools setup stage. C_n denotes the corresponding results from the commit under consideration. The comparison is made with respect to the results produced when reproducing the regression. This means that they will have been produced using the same microbenchmark on the same machine under similar circumstances, which makes the results less susceptible to variations due to outside factors compared to the previous results from the database.

If the condition isn't met, the assumption is made that regression is introduced at some point after the commit.

5.3.3 Building firmware

In the platform repository, the tools needed to build firmware are included. The wanted commit can be built using Bitbake once it has been "checked out". Provided with the specified product for which to build firmware, the building process is automatic and results in a firmware image that can be loaded into the product. The process of building the firmware is slow and can sometimes result in an error. When an error occurs, the tool automatically changes to an adjacent commit if an error is encountered when building firmware. Figure 5.3 depicts the principle components for this process. A brief presentation of Bitbake is included in **Appendix B**

5.3.4 Stopping condition

Once the search has nowhere to branch, that is, when two adjacent commits have been benchmarked, and the regression is present in the latter commit and not the former, the stopping condition has been met. The figure 5.2 shows an example of this scenario, where the results

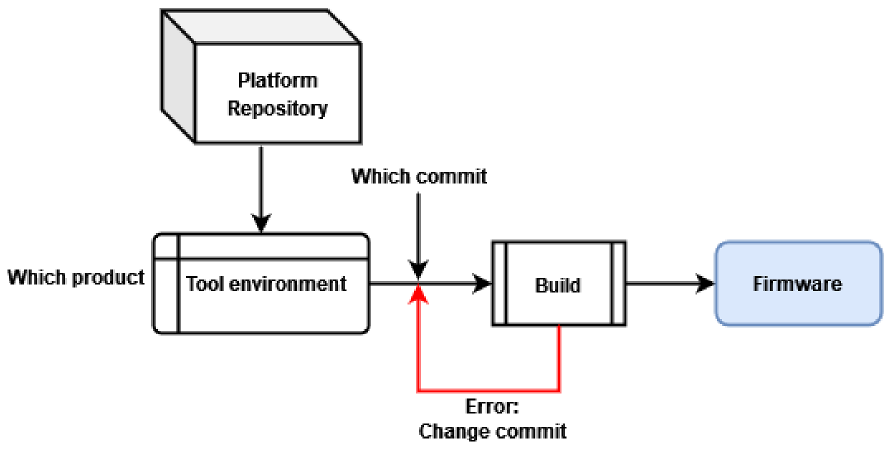

Figure 5.3: For the purposes of this thesis, the needed components for building new firmware are a suitable environment, a commit from which to build and the intended product the firmware is for.

for commit *A*-4 and *A*-5 indicate that the stopping condition has been met. Since *A*-5 is the earliest commit where the regression has been detected, this commit is concluded to be responsible for its introduction.

Additionally, edge cases such as when building errors are present between commits where the results indicate that the regression has been introduced somewhere in between also stop any further search. If building errors have been produced with every build between two commits where the results indicate that the change has been introduced, the process also stops.

5.3.5 Result

Information needed during the process of running the tool, such as which metric to evaluate, which benchmark scenario configuration to execute, the results from the benchmarks, and commit logs, are all stored in an object shared by the different components of the tool. This information, together with the outputs of the builds and benchmarks, is available in real-time during the execution. The shared data object and the resulting identification is outputted and uploaded to a database once the process is finished. This way, the results that the identification is based on can be inspected and evaluated both during and after the process is finished.

Chapter 6
Evaluation

Evaluation refers to the activity of empirically validating the implemented solution in its intended context. The primary goal of empirical validation is to assess whether the proposed solution is feasible for the given problem. For this thesis, that meant executing an implementation of the proposed solution on a selection of observed regressions, and for each evaluate whether the tool is applicable and whether the identified cause seems correct. Further, the evaluation seeks to asses which cases the tool could and could not be used, as well as analyzing its time consumption.

6.1 Data collection

The data presented in this chapter was the resulting data produced by an implementation based on the design described in chapter 5. The set of input variables used, *version, product, and metric*, were extracted from the backlog of known performance regressions. Each element of the backlog consisted of one, or a set of products, which version and in what performance metric the change in performance was observed. The same regressions can often be detected in groups of products, ranging between anything from two to as many as 20 or 30. This is to be expected since many products share functionality and hardware. However, the cause of these regressions are usually the same, and including large amounts of, essentially, the same data was deemed both unnecessary and potentially skewing of the data.

For this reason, only one of the products for any observed regression was included to avoid redundancy in the presented data unless the results diverged significantly. The usable subset from the backlog was further reduced by the lack of availability of old firmware in the database and the lack of specific products in the product pool utilized for the project.

With these constraints in consideration, a set of 37 separate executions of the tool were included as the basis for evaluation. To evaluate the performance of these executions, the relevant benchmarking results and their reference results are presented as plots. Since none of these observed regressions had a previously identified cause, the evaluation of the tools' ability to identify causes had to be made on a case-by-case basis, reviewing and interpreting the plots manually.

The time consumption for the different stages of each execution was also gathered to better assess whether the method is useful in a practical context. The stages of focus are; building firmware, performing benchmarks, and connection setup.

6.2 Execution results

Out of the 37 executions, 22 identified a cause for the regression. For one it was unable to pinpoint a definitive commit due to a building error, and the remaining 14 executions were unable to reproduce the regression which had previously been observed for the given set of circumstances. Utilizing these statistics as a basis for evaluation is not very helpful, however, since the identified causes have not been validated.

To concisely present the results of the evaluation, each execution has been placed in one of three categories. The categories are *successful, non-reproducible* and *inconclusive*. The categorizations were made through inspecting and judging the plots manually one by one. The category of unclear results includes the cases where several executions using the same parameters produced divergent results. Since the categories *successful* and *non-reproducible* make up the majority of the results, only a subset of the executions from these categories will be presented in this chapter. Table 6.1 contains the partitioning of executions belonging to each category.

Category	Number of Executions
Successful	19
Non-reproducible	13
Inconclusive	5
Total	**37**

Table 6.1: The amount of executions corresponding to each category.

6.2.1 Successful results

The results from a subset of the executions placed in the Successful category are featured in figure 6.1. Each plot shows the benchmarking results for the specified metric. The left- and rightmost data points correspond to the firmware available from the daily builds. The indices between them correspond to the bisected commits, each marked with a number specifying its placement in the version control log. The identification is deemed reliable as long as the measurements do not show excessive signs of noise and variation.

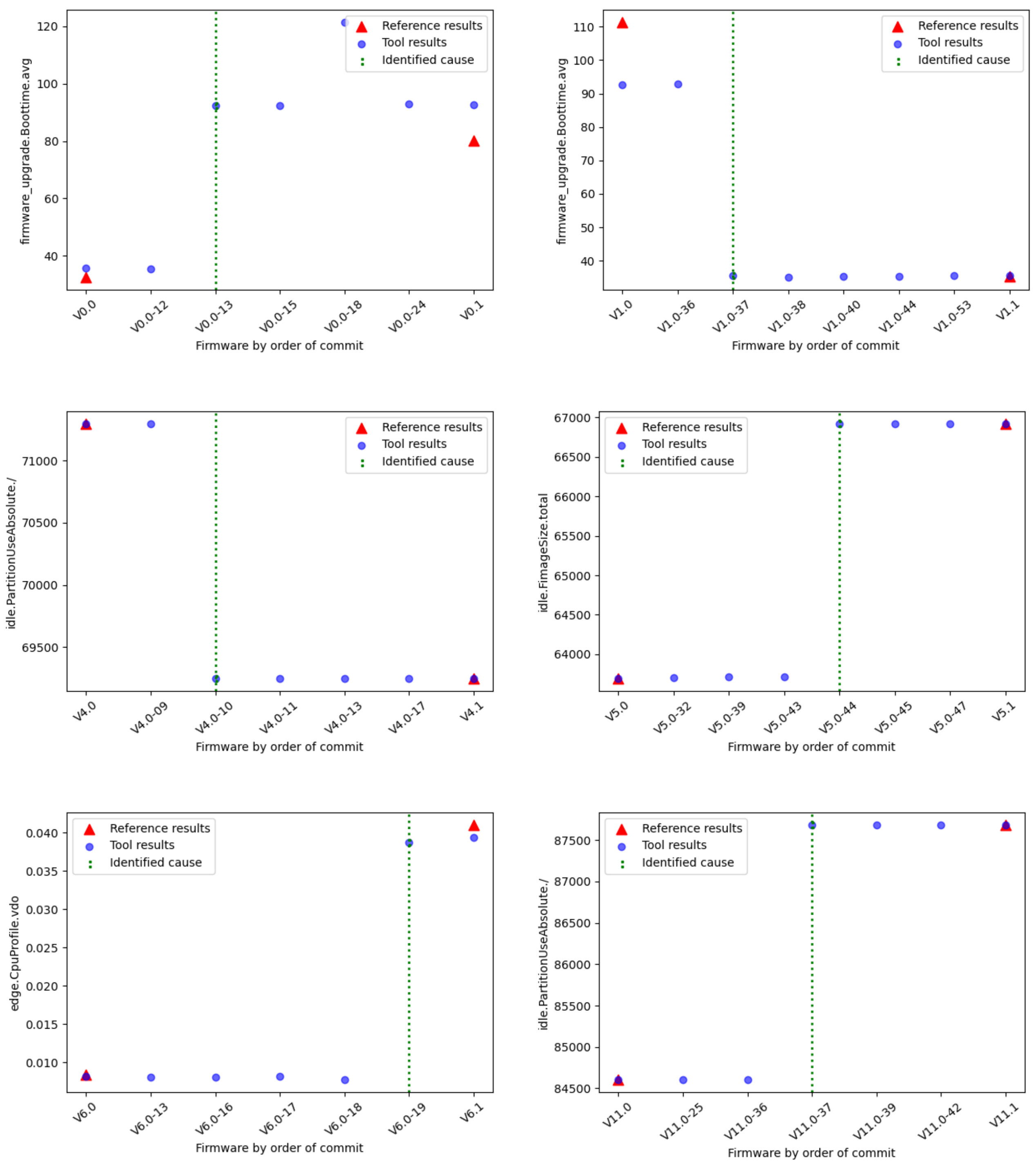

Figure 6.1: Examples of executions where a successful identification was made.

6.2.2 Non-reproducible results

Some executions were unable to produce similar results to the reference results for which the regressions had been identified previously. A subset of the executions from this category are included in the figure 6.2. In the last plot of figure 6.2 the regression is not reproduced, the only such instance in the executions used for evaluation. For the remaining executions, the regression was present in the firmware which it previously had not been. This indicates that the regression had been introduced externally, to anything from the benchmarking software, to the infrastructure, the network, or the units themselves.

However, getting non-reproducible results was expected since not only changes to the platform impact the benchmarking results. The inability to reproduce a regression can also be informative regarding the cause since the changes to the platform can, in most cases, be ruled out as potential causes. Although the cause of these regressions remains unidentified, the ability to determine what they are is outside the scope of this project.

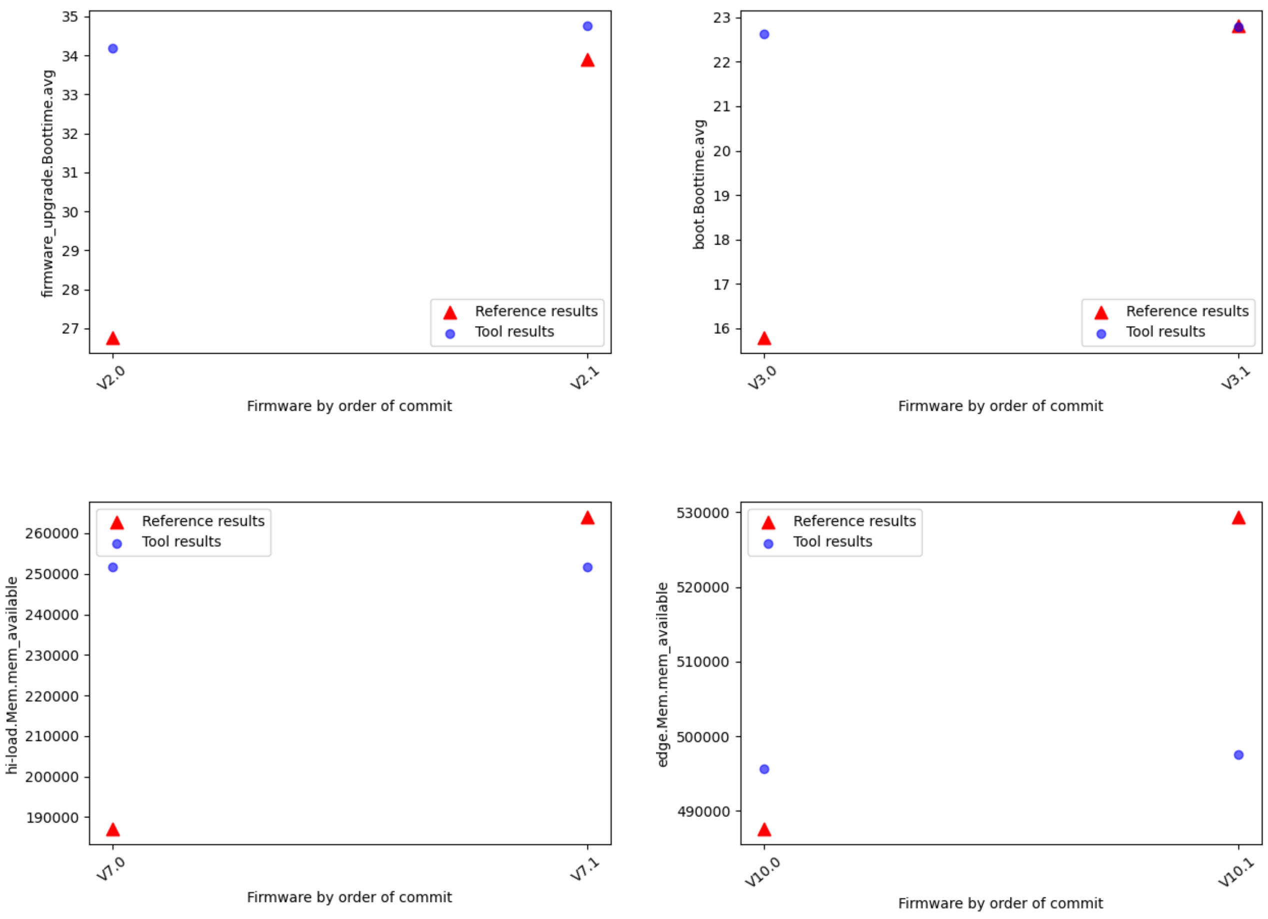

Figure 6.2: Examples of executions where the tool was unable to produce results similar to the reference results.

6.2.3 Inconclusive results

The executions that produced ambiguous results were placed in this category. The results from these are featured in figures 6.3, 6.4 and 6.5. The examples included in this category highlight some of the problems with the current implementation, which are important to keep in mind for any future use of a tool of this nature.

The execution featured in figure 6.3 was unable to pinpoint a specific commit due to a building error associated with one of the potential points of introduction. The building error meant that the corresponding benchmark could not be performed. Since the concerned commit, in this case, is the last and both the adjacent commit had been benchmarked earlier, the execution stops. In practice, this meant that the tool narrowed down the potential regression causes to the two marked in the plot.

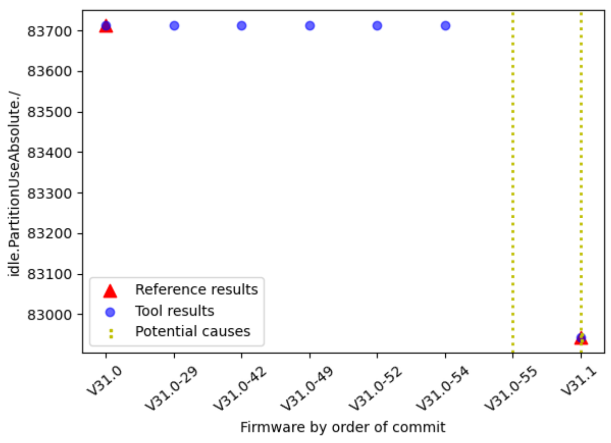

Figure 6.3: An example of an execution where a building error made a conclusive identification impossible.

Figure 6.4 includes the results from two separate executions of the same regression instance. Each execution resulted in a different identified point of introduction where neither was deemed sufficiently convincing given the variations in the benchmark results. Noisy metrics indicate that more measurements would be needed, from which a statistically informed identification could be made. This functionality is, however, not incorporated in the solution and would likely be extremely time-consuming. Moreover, the regression could be caused by any number of combinations of commits which would render any specific identification impossible to make.

Another case involving two diverging results is included in figure 6.5. Similar to the previous example, the variations in the benchmarks results are great enough to alter the course of the execution.

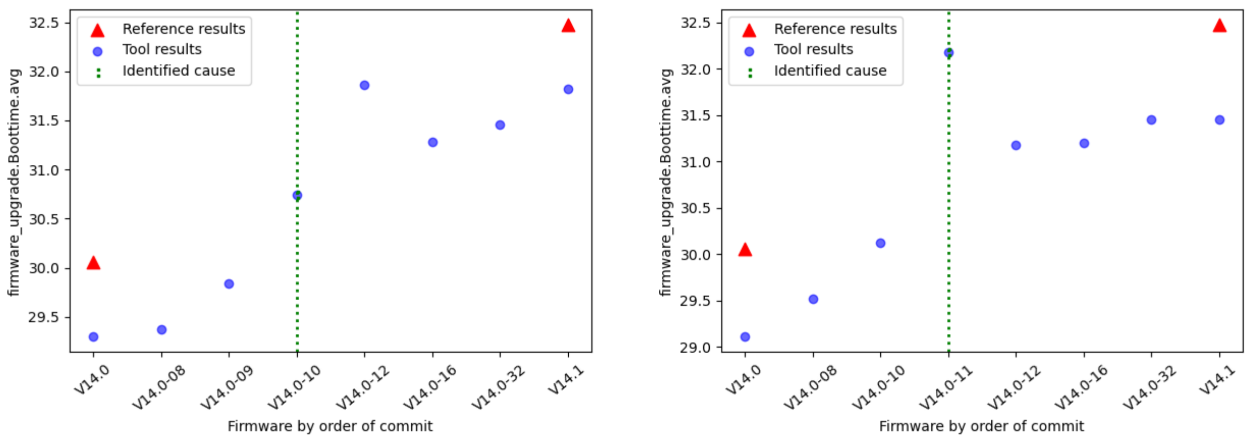

Figure 6.4: A case where two separate executions gave different results, both of which unreliable.

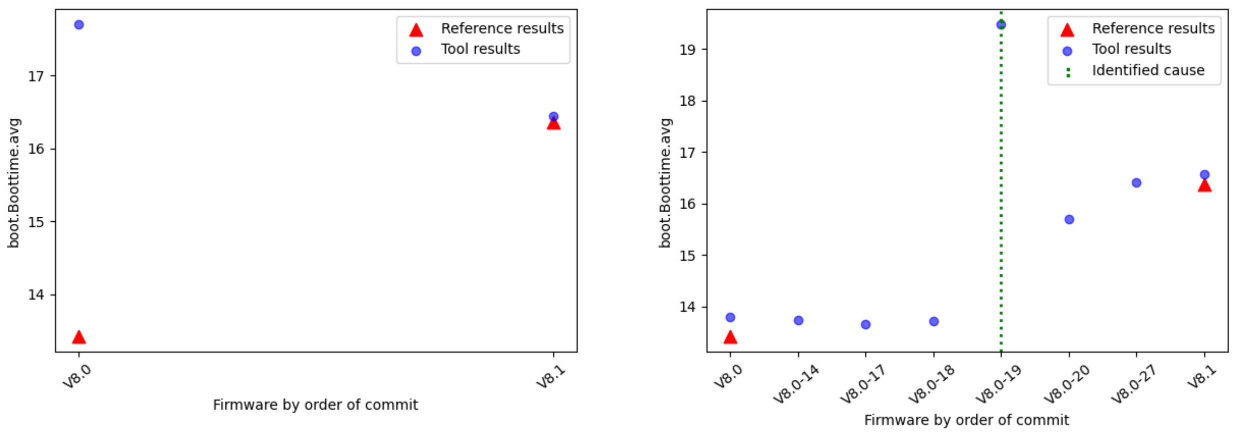

Figure 6.5: Another case where two separate executions gave different results.

6.3 Resource consumption

To assess whether the tool is feasible to use in a practical context, the time consumption for the different stages of each execution was gathered. This is highly relevant since one of the main concerns at Axis regarding their benchmarking practices is to improve the rate and efficiency of the feedback to the developers. The regression cause identification process is a time-consuming activity, whether it be manual or automatic. A box-plot of the time spent executing each finished execution (with a resulting identification) is shown in figure 6.6. Visible in the plot is that the execution times vary dramatically, with an average of about 2 hours and 42 minutes.

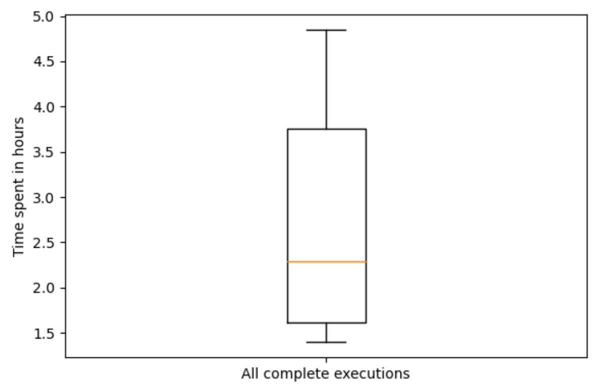

Figure 6.6: Box-plot distribution of the execution time of all finished executions.

To better understand the variations in time consumption, individual components were monitored. More specifically, the time spent benchmarking, building firmware, and connecting to the unit. The connection process involves establishing communication to the unit, loading it with the intended firmware, and preparing it for the following benchmarks. This process is repeated with every benchmark. A box-plot of the time spent connecting is shown in the figure 6.7. The average time spent connecting to a unit was 2 minutes and 42 seconds.

Scenario	**Average Time Consumption** (min)	**Number of Cases**
Boot-time	11:47	3
Idle	2:51	12
Edge	5:50	6
High-Load	10:23	3
Rtsp-Response	4:56	2
Firmware-Upgrade	14:53	4
Mjpeg	3:43	2

Table 6.2: Average time consumption for each microbenchmark utilized. Note that one case would correspond to either 2 or more benchmarks depending on reproducibility.

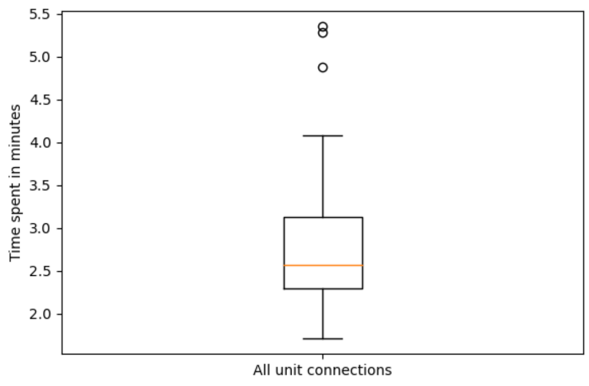

Figure 6.7: Box-plot distribution of all the unit connection times.

Since the microbenchmarks utilized is one of the main components which would vary across executions, it was deemed more helpful to separate these. Table 6.2 shows the average time consumption for each utilized microbenchmark.

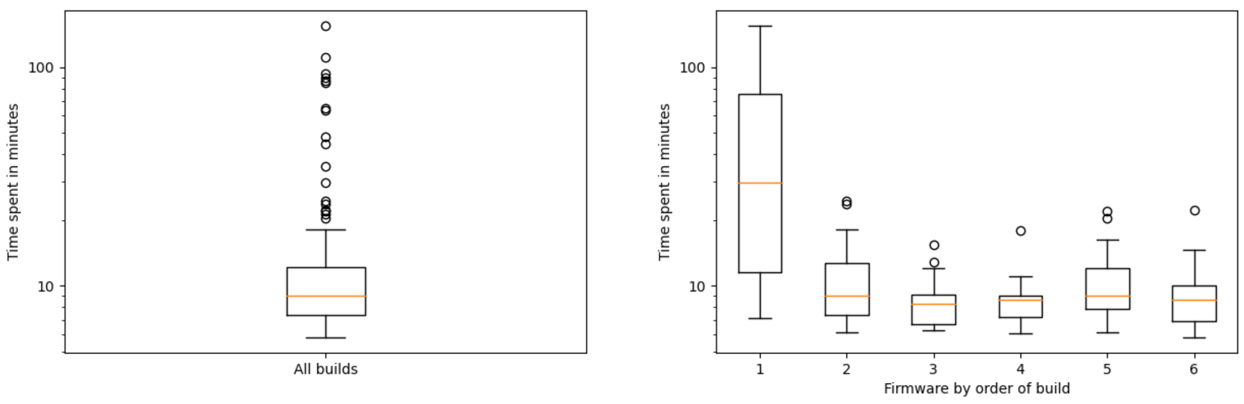

Figure 6.8: Box-plot distribution of all the build times.

In figure 6.8 box-plots of the time spent building firmware is presented. The left plot shows the time distribution of all the builds while the right one shows the distribution associated with each chronological build for an execution. The first firmware build usually consumes the most time since it involves setting up the building environment needed to produce firmware. The following builds appear to have very similar distributions. The average time consumed building firmware was 15 minutes and 51 seconds, while for the first build it was 31 minutes, and the rest 9 minutes.

The components included in the time consumption evaluation have significant variations in their results. Some of the discrepancies for firmware builds are likely associated with which products model it was built for. However, the lack of execution data makes any rigorous evaluation of this difficult, within the scope of this thesis. Additionally, the build time measurements were conducted in such a way that the occasional build errors are incorporated into the build time of the next commit which was built successfully. The two factors mentioned above are likely significant contributors to the deviations in build time data.

6.4 A model for time estimation

Execution time varies greatly depending on the target microbenchmark scenario, which makes the average time consumption can be misleading as a reference for future executions. Further, the data is skewed towards certain scenarios, as is shown in table 6.2. Since it is of interest to better estimate the time consumption of an execution before it is made, a simple model can be derived from treating the components of the execution separately. The model can be expressed as follows:

$$(time_{bm} + time_c) \times number_{bm} + time_{bd} \times number_{bd} = time_{tot} \tag{6.1}$$

Where $time_{bm}$ denotes the execution time for the given benchmark, $time_c$ the connection time and $number_{bm}$ the estimated number of benchmarks needed to finish. $time_{bd}$ denote the firmware build time and $number_{bd}$ the estimated number of builds. Using the averages presented in the previous section, it is possible to estimate the time consumption based on the gathered data. The average number of commits to the platform for any given version is 54, in which case, six bisections are needed to find the cause. That means eight benchmarks are needed, including the initial two.

Utilizing (6.1) with the averages presented in the previous section to estimate the time consumption for an execution involving the Idle scenario gives $(2.85+2.70)\times 8+15.85\times 6 \approx 2$ **hours and 20 minutes**. Likewise, an estimation for the Firmware-Upgrade scenario gives $(14.89 + 2.70) \times 8 + 15.85 \times 6 \approx 3$ **hours and 56 minutes**.

6.5 Results from the evaluation

The evaluation has been aimed at empirically validating the implemented solution in its intended context. Execution data was gathered to conclude whether the proposed solution is feasible for the given problem. Although the amount of useful execution data was found to be limited, enough was gathered to make an overall assessment of the tool. Table 6.1 summarizes the execution results from the evaluation of the tool in terms of its intended use. In 33 out of the 37 executions, the tool performed its role to the extent which could be expected. These include the executions put in the successful and non-reproducible categories along with the inconclusive execution featured in figure 6.3 since the results produced were the most specific possible under the circumstances.

Chapter 7
Discussion

This chapter serves as an overall reflection of the results presented in the previous chapter, along with a discussion of the various biases, and threats to validity present in both the data collection and the solution as a whole. The final section also explores some thoughts regarding the future development possibilities of the project.

7.1 Conclusions from the evaluation

The solution showed itself to be potentially useful, providing a seemingly accurate result in 19 out of the 37 performed executions. In 13 of the executions the tool was not applicable but could be informative in a wider search for regression causes. The remaining five executions consisted of cases where the identifications made were inconclusive. Overall the tool was found to perform its role to the expected extent in 33 out of the 37 executions, given that the non-reproducible results and building errors could not be avoided under the circumstances.

As previously stated in chapter 4, the regular benchmarking practices at Axis involve roughly 1 hour and 30 minutes for approximately 100 products three times a day, five times a week. This adds up to $1.5 \times 3 \times 5 \times 100 = 2250$ **hours** of benchmarks a week. The frequency of regression detection is about three cases a week according to the interviews. If the tool used for each of these, the average time consumption would be $2.72 \times 3 = 8.16$ **hours** per week if all were successful, and less if some of the cases were non-reproducible. This would make the overall resource consumption for using the tool in these cases an increase of $8.16 \div 2250 = 0.36\%$, which is not a significant increase.

As shown by the examples in figures 6.4 and 6.5, executions with the same set of parameters can give vastly different results. These examples show how variation in the results can alter the course and classification of the execution. Since these variations could be present in any metric, the results from this tool cannot always be taken at face value and should be interpreted by an expert before being acted upon. Some metrics are much more prone to these variations than others which means that a person with expertise should be able to evaluate these cases with sufficient skepticism.

The executions in 6.4 showed an indication of having gradual increases in their results. Although difficult to conclude in this case, it highlights the potential issue of gradual regression introductions. If a regression is introduced gradually over several commits to one version the solution presented would not be able to pinpoint these introductions sufficiently. Only the first result above a certain threshold would be identified. The insights gained from the examples in figure 6.5 challenged some of the seemingly obvious categorizations of cases with non-reproducible results. Most of the cases of non-reproducible results included in the evaluation had been executed several times to address this issue. For any potential future use, this issue should be kept in mind.

Benchmarking and firmware building are time-consuming processes. For the tool to reach a conclusion, it needs to perform both of these activities repeatedly. This made time consumption one of the main bottlenecks for any further development of the proposed solution. Had it been able to produce firmware or benchmark results faster, statistical methods could be utilized to better evaluate inconclusive results. For most cases, however, the simple threshold solution proved sufficient.

7.2 The tool in practice

Towards the later stages of the project at Axis, the tool was put into use for the regressions detected in their daily benchmarks. The execution of version V11.0 to V11.1 included in figure 6.1 is an example of an identification made in the intended practical context of the work at Axis. The execution of version V2.0 to V2.1 in figure 6.2 is another example. Since then it has been put into regular use by the platform coordinator, whose job it is the evaluate the benchmarking results and maintain a dialog with the developers.

The overall lengthy time consumption for the tool is one of its drawbacks. It is, however, less time-consuming than performing the same process manually and the ease with which it can be used makes it a potentially useful complement to the practices in place at Axis. The resources needed for it to run are an automation server index and a unit to perform benchmarks on, both of which are available in abundance at Axis.

An additional resource needed to keep the tool useful at Axis is overall maintenance. Like any piece of software which is a part of a larger system, regular maintenance and updates are likely to be needed in case changes were to be made to the systems that the tool utilizes.

The evaluation showed that the tool was able to function in its intended context much like it was intended. The criteria from the **Project specification** were met and the overall resource consumption was not found to be excessive. Thus, to address **RQ3**, I would deem it feasible to use in the context which it has tested.

7.3 Data selection

The work behind chapter 6 on evaluation was an attempt to assess the validity of the solution in a practical context. The basis of evaluation did, however, prove to be more limited than first expected, since the number of usable cases was reduced by factors such as availability of firmware and product units. Furthermore, the observed regressions in the backlog had no explicitly documented cause, making it difficult to assess the validity of the results of the executions.

The subset of regression evaluated could also be subject to certain biases. The method used for detecting regressions was a manual inspection of plots, often made up of averages over various products. This approach introduces certain biases which ultimately have influenced the basis of evaluation for this thesis. Furthermore, the limited time frame of the regressions evaluated may also be subject to biases regarding which aspects of the various firmware were being developed at that time.

7.4 Future work

Although the presented solution proved itself potentially useful there are many ways in which it could be improved. The tool could incorporate further automation, such as, anomaly detection of the daily benchmarking results. Another thesis conducted by Dageson and Hedesand explores various methods of anomaly detection in this context at Axis. [14] If combined successfully, this would further minimize manual human intervention, and the tool could be executed in conjunction with the benchmarks, thus, producing results earlier in the day.

The current design of the tool takes no account of which metric it evaluates other than in its choice of microbenchmark to match it. However, some metrics are the product of an average taken from several measurements. These metrics are usually the most susceptible to noise and variation and have a corresponding standard deviation metric available. This means that a possible way to address the issues raised with inconclusive results due to variations is to incorporate the corresponding metrics in their assessment.

To improve efficiency, there could be ways to filter the commits to the platform, reducing the number of potential points of regression introduction. However, this would require extensive knowledge of the platforms' various source code components, along with its utilized packages and libraries. Hence, the usefulness of this approach is highly speculative but potentially worthy of exploration.

Another way to potentially improve time efficiency is to parallelize the execution of the benchmarks and the building of firmware. This approach was not pursued during the development of the tool since the branching direction could not be assessed until the benchmark was finished. Hence, two potential firmware would have to be built for each bisection where only one was utilized. If parallelization was found to decrease time consumption significantly, however, it could be pursued.

If greater time efficiency was be achieved or is deemed less relevant, repeating benchmarks to incorporate analysis of statistical distributions of the results is an option to improve the rigor of the benchmark result assessment of the tool. Especially if coupled with the approach of utilizing related metrics (averages and standard deviations).

Chapter 8
Conclusion

The work of this thesis has been a case study investigating a method of identifying the causes of performance regressions in the context of large-scale benchmarking of in-development firmware at Axis Communications. Utilizing a design scientific approach, the project set out to assess how the current practices at Axis were conducted and where its main concerns were. A solution addressing the relevant challenges was then developed and subsequently evaluated in its intended context.

The proposed solution utilizes the version control logs of their software platform, firmware build- and benchmarking tools for their products to pinpoint regression introducing commits. The method used sought to minimize human intervention in the regression cause identification process. An evaluation assessed that the proposed solution was successful in identifying the causes of more than half of the regressions it was tested on, where the unsuccessful cases were mostly due to an inability to reproduce the previous regression results.

Although time-consuming to execute, the overall resource consumption was found to be negligible compared to the overall benchmarking currently in practice. New regressions are detected each week and the process of diagnosing them can be tedious. Therefore, the tool developed was found feasible to use for the process of identifying regression causes when these cases come up in the day-to-day workings at Axis.

References

[1] A brief overview of jira. https://www.atlassian.com/software/jira/guides/getting-started/overview. (Accessed 2021-11-23).

[2] Getting started: The yocto project® overview. https://www.yoctoproject.org/software-overview/. (Accessed 2022-01-7).

[3] Infrastructure. https://www.jenkins.io/projects/infrastructure/. (Accessed 2022-01-7).

[4] Jenkins. https://www.jenkins.io/. (Accessed 2022-01-7).

[5] Pipeline as code with jenkins. https://www.jenkins.io/solutions/pipeline/. (Accessed 2022-01-7).

[6] What is a container? https://www.docker.com/resources/what-container. (Accessed 2022-01-7).

[7] ISO/IEC 25010. Systems and software engineering—systems and software quality requirements and evaluation (square)—system and software quality models. 2011.

[8] Scott Chacon and Ben Straub. *Git Tools*, pages 181–277. Apress, Berkeley, CA, 2014.

[9] Jinfu Chen. Performance regression detection in devops. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 206–209, 2020.

[10] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, 2017.

[11] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. The use of change point detection to identify software performance regressions in a continuous integration system. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, page 67–75, New York, NY, USA, 2020. Association for Computing Machinery.

[12] N. Deepa, B. Prabadevi, L.B. Krithika, and B. Deepa. An analysis on version control systems. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–9, 2020.

[13] Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25:2630, 2020.

[14] André Hedesand and Oliver Dageson. Abnormality detection in diagnostics data from network cameras. Master's Theses in Mathematical Sciences. Lund University, 2021. Student Paper. `https://lup.lub.lu.se/student-papers/search/publication/9068634`.

[15] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 27–38, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Generic environment for full automation of benchmarking. In *Proceedings of the 1st International Workshop on Software Quality (SOQUA)*, pages 182–196. GI, September 2004.

[17] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Benchmarking Basics*, pages 3–21. Springer International Publishing, Cham, 2020.

[18] David B. Leblang and Paul H. Levine. Software configuration management: Why is it needed and what should it do? In Jacky Estublier, editor, *Software Configuration Management*, pages 53–60, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[19] Nasir Mehmood Minhas, Kai Petersen, Jürgen Börstler, and Krzysztof Wnuk. Regression testing for large-scale embedded software development – exploring the state of practice. *Information and Software Technology*, 120:106254, 2020.

[20] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. Perfranker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 23–34. Association for Computing Machinery, 2017.

[21] Colin Robson. *Real world research : a resource for social scientists and practitioner-researchers.* Blackwell, 2002.

[22] Per Runeson, Emelie Engström, and Margaret-Anne Storey. The design science paradigm as a frame for empirical software engineering. In *Contemporary Empirical Methods in Software Engineering*, pages 127–147, Germany, 2020. Springer.

[23] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.

[24] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. Performance evolution matrix: Visualizing performance variations along software versions. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 1–11, 2019.

[25] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, page 15–26, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Gary Stringham. Chapter 1 - introduction. In Gary Stringham, editor, *Hardware/Firmware Interface Design*, pages 1–18. Newnes, Boston, 2010.

[27] David V. Thiel. *Literature search and review*, page 27–72. Cambridge University Press, 2014.

[28] Joan E van Aken. Management research based on the paradigm of the design sciences: the quest for field-tested and grounded technological rules. *Journal of management studies*, 41(2):219–246, 2004.

Appendices

Appendix A
Interview Questions

1. What is your title and what is your relation to the current internal benchmarking? What does your role entail?

2. Are there other major benchmarking or performance testing going on at Axis? Is there any communication there?

3. What has been your previous approaches in this area and how long has the workflow been how it is today?

4. How frequently do the benchmarks reveal a performance issue?

5. What does the current system of issue tracking look like?

6. Who are the main stakeholders of the daily internal benchmarking?

7. What does the current dialog with the stakeholder look like?

8. What does the current dialog with the development teams look like? Would a tool like this improve this in any way?

9. Whats your expectation of firmware bisection and do you expect to utilize these types of solutions more in the future? Would a tool like this effect the dialog with the development teams?

10. Do you have an idea of how to more rigorously evaluate the overall performance of a tool like this?

11. Do you have any useful advice or information you would like to share which was not covered by these questions?

Appendix B
Tools

B.1 Git

Git is an advanced version control system created in 2005 and has got wide extensive support of the open-source community, consisting of quality developers behind. Git provides easy control over changes made to source code in both linear and nonlinear software development. Git supports branching, undoing, merging, among other things and is by far the most popular version control system available as of writing. Git has a distributed repository type with cryptographic integrity where every file and commit is checksummed documented and rebuildable. [12]

In addition to being primarily for version control, Git also includes features to provide debugging functionalities to source code projects. Since Git is designed to handle nearly any type of content, these tools are fairly generic, but they can often aid bug identification or culprit when or where things went wrong. [8]

B.1.1 Git bisect

Git bisect is a command that performs a binary search through the commit history of a repository to help with identification of which commit introduced an issue. It is used by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then git bisect picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the issue. [8]

The bisection process used in this thesis is however implemented manually to more easily log useful information. The same principles apply however.

B.2 Docker

Docker is a set of platform as a service products that use operating system level virtualization which allows software to be delivered in packages which are called containers. A container is a standard unit of software that packages up code and all its dependencies so that an application can run quickly and reliably from one computing environment to another. A Docker container image is a small, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

When an image is executed a container is built and run according to the image specification. Containerized software ensures that it always will run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging. Docker is available for both Linux and Windows-based applications. [6]

B.3 Jenkins

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and deploying software. Jenkins can be installed through native system packages, Docker, or run standalone by any machine with a Java Runtime Environment installed and provides hundreds of plugins to support building, deploying and automating any project. Often used as a simple CI server or turned into the continuous delivery hub for any project, it can easily distribute work across multiple machines, helping drive builds, tests and deployments across multiple platforms faster. [4]

As an independent open source project, Jenkins maintains most of its own infrastructure including services which help keep the project running. This includes anything from operating virtual machines and distribution networks, to project-specific applications developed to make the development of Jenkins core and plugins more efficient. [3]

The default and most common interaction model with Jenkins, is web UI driven, requiring users to manually create jobs, then manually fill in the details through a web browser. This allows the user to create and manage jobs to test and build multiple projects. It also keeps the configuration of a job to build, test and deploy separate from the actual code being built, tested and deployed. [5]

B.4 Yocto

The Yocto Project is an open source project which aim is to help developers make custom Linux-based systems for embedded products, regardless of hardware architecture. The project provides a set of tools and an ecosystem where technologies, software stacks, configurations and best practices used to create tailored Linux images for embedded devices, can be shared. Further, the project contributes a standard to delivering software stacks and hardware support which allows for the interchange of software configurations and builds. The tools can be used to build and support customizations for multiple hardware platforms and software stacks in a scalable and maintainable way. The project originated from, and works with the OpenEmbedded Project which is where some of the meta-data and its build system are derived from. [2]

Yocto at its core combines, maintains and validates the three following key development elements:

1. A set of integrated tools to make working with embedded Linux successful, such as automated building and testing, processes for board support and license compliance, as well as component information for custom embedded Linux-based systems
2. A reference embedded distribution
3. The OpenEmbedded build system, co-maintained with the OpenEmbedded Project

B.5 Bitbake

Bitbake is the build system which the Yocto project utilizes. A scheduler and execution engine which parses recipes and configuration data for a custom Linux distribution builds. BitBake recipes specify how a particular package is built. They include all the package dependencies, source code locations, configuration, compilation, build, install and remove instructions as well as the metadata for the package in standard variables. Related recipes are consolidated into a layer.

Bitbake then creates a dependency tree to order the compilation, schedules the compilation of the included code. Ulimately the building of specified Linux image is executed, where an image is a binary form of a Linux distribution intended to be loaded onto an embedded device. During the build process dependencies are tracked and native or cross-compilation of the package is performed. As a first step in a cross-build setup, the framework will attempt to create a cross-compiler toolchain suited for the target platform. [2]

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-02-17

EXAMENSARBETE Optimizing regression benchmarking for network video products
STUDENT Thomas Rodenberg
HANDLEDARE Masoumeh Taromirad (LTH)
EXAMINATOR Per Runeson (LTH)

Identifiera orsaken till prestandaförändringar

POPULÄRVETENSKAPLIG SAMMANFATTNING **Thomas Rodenberg**

För att kunna upprätthålla god prestanda i ständigt uppdaterade kodbaser behövs regelbunden prestandatestning. När en förändring upptäcks vill man identifiera dess orsak så snabbt och effektivt som möjligt. Detta arbete har fokuserat på att utveckla ett verktyg som utför denna identifiering automatiskt.

Axis Communications är en ledande utvecklare av olika nätverksprodukter med ett huvudfokus på övervakningskameror. Mjukvaran till deras produkter uppdateras dagligen varvid omfattande tester görs för att säkerställa att mjukvarans prestanda inte försämrats genom ändringarna. Prestandatestningen omfattar mätningar för egenskaper såsom CPU- och minnesanvändning för mer än 100 olika produkter. Upptäcks en prestandaförsämring (regression) för någon av dessa krävs det att man kan identifiera var den introducerats för att kunna åtgärda problemet. Varje daglig mjukvaruuppdatering består av en samling mindre deluppdateringar från en mängd olika utvecklingsgrupper på företaget. Då arbetet att identifiera vilken deluppdatering som orsakat en försämring ofta kan vara krånglig och tidskrävande vill man kunna automatisera denna process så mycket som möjligt.

Prestandatestningen är mycket tidsomfattande men kan delas upp i korta delmoment som fokuserar på en delmängd av alla de mätningar som normalt utförs. Verktyget som utvecklats kombinerar de nedkortade delmomenten från de befintliga prestandatesten med en sökning genom deluppdateringarna för att automatiskt kunna säkerställa var försämringar introducerats. Sökningen görs på baserat på resultaten från prestandatesten där man kan detektera en tydlig förändring för den drabbade egenskapen. När exekveringen är färdig skall verktyget kunna återge vilken deluppdatering som är skyldig till försämringen, förutsatt att den kunnat återskapas.

Som utvärdering på arbetet sattes verktyget i praktisk användning vid både nyupptäckta försämringar och ett urval av de tidigare noterade försämringarna som fanns tillgängliga. Bristande tillgänglighet av tidigare bekräftade orsaker gjorde att utvärderingen av verktygets identifiering fick göras manuellt. Verktygets resursåtgång utvärderades också som underlag för huruvida en framtida användning och utveckling var relevant. Den praktiska utvärderingen kunde fastställa att verktyget var framgångsrikt i majoriteten av de testade fallen samt visade potential att kunna användas och vidareutvecklas på Axis.