DEGREE PROJECT IN COMPUTER SCIENCE 120 CREDITS, SECOND CYCLE

STOCKHOLM, SWEDEN 2016

Optimizing Ruby on Rails for performance and scalability

KIM PERSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Optimizing Ruby on Rails for performance and scalability

Optimering av Ruby on Rails för prestanda och skalbarhet

KIM PERSSON
KIMPE@KTH.SE

Degree project in Computer Science
Master's Programme Computer Science
Supervisor: Stefano Markidis
Examiner: Jens Lagergren
Employer: Slagkryssaren AB
February 2016

Abstract

Web applications are becoming more and more popular as the boundaries of what can be done in a browser are pushed forward. Ruby on Rails is a very popular web application framework for the Ruby programming language. Ruby on Rails allows for rapid prototyping and development of web applications but it suffers from performance problems with large scale applications.

This thesis focuses on optimization of a Ruby on Rails application to improve its performance. We performed three optimizations to a benchmark application that was developed for this project. First, we removed unnecessary modules from Ruby on Rails and optimized the database interaction with Active Record which is the default object relational mapping (ORM) in Ruby on Rails. It allows us to fetch and store models in the database without having to write database specific query code. These optimizations resulted in up to 36% decrease in application response time. Second, we implemented a caching mechanism for JavaScript Object Notation (JSON) representation of models in the optimized application. This second optimization resulted in a total of 96% response time decrease for one of the benchmarks. However, we also observed an increased response time (with approximately with 15%) for a second benchmark compared to the unoptimized application. Third, we tuned the garbage collector for CRuby. CRuby is the original implementation of Ruby created by the inventor of the language Yukihiro Matsumoto. This optimization was performed on top of the already optimized application with caching and resulted in an overall decrease in response time: one benchmark showed a response time decrease of 29% with regard to the unoptimized application.

In addition, we evaluated the performance of different Ruby implementations: we studied the performance of the application with CRuby, JRuby and Rubinius. We observed that JRuby had the best performance. When the optimized and cached application executed in JRuby it obtained a decrease in response time of up to 32% with regard to the unoptimized application in CRuby.

In conclusion, we presented that different optimization techniques can be applied to improve Ruby on Rails performance leading to maximum performance increase of 96%. In addition, we showed that the choice of Ruby implementation is very important as the JRuby implementation has potential to further increase application performance. This work is very important for future development and success of the Ruby on Rails framework as performance is largely seen as the biggest disadvantage with the framework.

Referat

Optimering av Ruby on Rails för prestanda och skalbarhet

Webbapplikationer ökar i popularitet allt eftersom gränserna för vad som går att göra i en webbläsare flyttas framåt. Ruby on Rails är ett populärt ramverk för utveckling av webbapplikationer med programmeringsspråket Ruby. Ruby on Rails gör det möjlig att snabbt utveckla en produkt men ramverket lider av prestandaproblem vid utveckling av storskaliga applikationer.

Detta examensarbete fokuserar på optimering av en Ruby on Rails applikation för att förbättra dess prestanda. Vi utför tre optimeringar på en utvärderingsapplikation som utvecklades för detta projekt. Först tog vi bort alla moduler moduler som inte behövdes från Ruby on Rails ramverket och optimerade Active Records interaktion med databasen. Active Record är Ruby on Rails objekt-relations-mappning (ORM). Ett ORM gör det möjligt att hämta och lagra information i databasen utan att behöva skriva databasspecifika anrop. Dessa optimeringar resulterade i en upp till 36% minskning i applikationens svarstid. I andra hand implementerade vi en caching modul för JavaScript Object Notation (JSON) representationer av databasmodeller i den redan optimerade applikationen. Denna optimering resulterade i en total minskning av svarstiden med 96% vid ett utvärderingstest. Vi såg dock en ökning av svarstiden (med uppskattningsvis 15%) för ett annat utvärderingstest jämfört med den icke optimerade applikationen. Den tredje optimeringen av applikationen vi utförde var att finjustera skräpsamlaren i CRuby. CRuby är den ursprungliga implementationen av Ruby som utvecklades av språkets skapare Yukihiro Matsumoto. Denna optimering utfördes ovanpå tidigare optimeringar och resulterade i en minskning i svarstiden för alla utvärderingstest: ett utvärderingstest visade en minskning av svarstiden med 29% jämfört med den icke optimerade applikationen.

Utöver ovan nämnda optimeringar utvärderade vi prestandan av olika Ruby implementationer: vi studerade prestandan för applikationen med CRuby, JRuby och Rubinius. We observerade att JRuby gav bäst prestanda. När den den optimerade applikationen med caching kördes i JRuby gav den en minskning i svarstid med upp till 32% jämfört med den icke optimerade applikationen i CRuby.

Avslutningsvis konstaterar vi att olika optimeringstekniker kan användas för öka prestandan av en Ruby on Rails applikation. Vi uppnådde en minskning av svarstiden med upp till 96%. Utöver detta visar vi att valet av Ruby implementation är mycket viktigt eftersom JRuby har potentialen att ytterligare förbättra en applikations prestanda. Detta arbete är viktigt för framtida utveckling och framgång för Ruby on Rails ramverket eftersom dess prestanda ofta är sett som den största nackdelen med att använda ramverket för att utveckla en webbapplikation.

Contents

Chapter 1

Introduction

Ruby on Rails [20] is a very popular web application framework for the Ruby [31] programming language. It has become a popular framework because of the developer productivity and the ability to be able to go from an idea to a working product in a very short time frame. Some developers who rely on Ruby on Rails to design a product will face performance problems as the user base increases beyond the scaling capabilities of the application. When this happens there are two main ways of solving the problem: either developers can try to optimize the application or they should rewrite it completely in another framework or programming language with better performance.

Previous work in this area generally takes a low level approach and focuses on how to improve the performance of Ruby by optimizing the interpreter [56], or eliminating the global interpreter lock through hardware transactional memory in order to achieve true parallelism. [53] This thesis takes a higher level approach and focuses on optimization of a Ruby on Rails application for better performance and scalability. We define scalability as how well the application can be scaled on a single server to support an increasing user base. This thesis does not consider scaling the application over multiple servers as this is handled outside of the Ruby on Rails framework.

In this thesis, we analyze tools and techniques to increase the performance of the Ruby on Rails framework. A Ruby on Rails application was developed to benchmark the performance improvements from different optimizations. The application is RESTful style API modeled after the popular image sharing social network Instagram. This application was used for benchmarking and measuring the effectiveness of the optimizations.

We perform three different optimizations to the application as well as evaluate the performance of different Ruby implementations. First we used the gem `Rails::API` to remove unnecessary modules from Ruby on Rails and optimized the database interaction with Active Record. Active Record is the default object relational mapping (ORM) in Ruby on Rails, it makes it easy to fetch and store models in the database without having to write database specific query code. Second, we

implemented caching for JSON representation of database models. Third, we tuned the garbage collector in Ruby using an automated analysis to try to find garbage collection parameters that improve the application performance.

In addition to the optimizations the application was evaluated with three different implementations of Ruby in order to conclude if performance gains can be achieved by switching to another Ruby implementation. We also discuss the performance of different Ruby web application servers and compare the performance of Ruby on Rails to the Ruby micro framework Cuba.

1.1 Scientific questions

The goals of this thesis is to analyze high level performance optimization of the Ruby on Rails framework. In order to maintain a scientific approach in trying to analyze the problem a set of scientific questions have been established. This thesis seeks to further research these topics and proposes answers to the questions.

- What tools can be used to evaluate Ruby on Rails performance and scalability?
- Does Ruby implementation impact performance?
- What parts of Ruby on Rails can be optimized?
  - Memory consumption
  - Object-relational mapping (ORM)
  - Garbage collection
  - Choice of application server
  - Removal of unnecessary modules

To answer these questions is very important for the future development of the Ruby on Rails framework. To show that there is a potential for a series of optimizations to improve the performance of the Ruby on Rails web applications.

1.2 Thesis overview

This thesis focuses on high level optimizations of Ruby on Rails. We optimize an existing Ruby on Rails application to improve performance and scaling on a single server. The purpose of this project is to investigate how much the performance of a Ruby on Rails application can be improved without having to make significant changes to the application code base.

Chapter 2 presents an overview of the background for the thesis, Ruby, JRuby and Rubinius are presented and performance related features of each of the different Ruby implementations are presented and analyzed. The chapter also gives an overview of the Ruby on Rails framework and describes why it is so popular in web

application development. Lastly the chapter describes different database technologies that can be used with Ruby on Rails as well as related works in this area of research.

Chapter 3 describes the methodology used for conducting the experiments of this project, it presents the different tools used for performance evaluation and benchmarking of the application. It also provides a detailed description of the environment in which the performance benchmarking and evaluation was performed.

Chapter 4 outlines the application that was implemented as a basis for optimization. It discusses the motivation for the choice of application and provides a detailed specification of the application functionality.

Chapter 5 presents different optimization techniques and tools. A subset of the techniques were chosen and performed as optimizations for this thesis project. The chapter also presents and motivates which parts of the application was used for benchmarking application performance.

Chapter 6 presents the results from the performed optimizations. It gives an overview of resulting application performance both in the form of average request response time and total system memory consumption. These metrics are used for drawing conclusions of the efficiency of optimization techniques for the Ruby on Rails framework.

Chapter 7 discusses the results from the optimizations and concludes whether the Ruby on Rails framework can be optimized for better performance and scalability. It proposes an optimization suite for Ruby on Rails applications that need better performance and more efficient scaling and suggests suitable topics for future work in this area of research.

Chapter 2

Background

In the previous chapter we presented an introduction to this thesis, this chapter provides background information. In this chapter we introduce three different Ruby implementations and discuss how they handle important performance aspects like multithreading and garbage collection. Unique aspects of each Ruby implementation are presented in a short and concise manner. An introduction to the Ruby on Rails is also provided as well as a short introduction to different database systems. Related works in the area of optimizing high level dynamic programming languages are presented together with works discussing relational and NoSQL databases and their performance.

2.1 Ruby

Ruby is an open source high level dynamic programming language created in 1995 by Yukihiro Matsumoto. It is an object oriented scripting language with focus on simplicity and programmer productivity. Ruby fully embraces the object oriented model, everything in Ruby is an object. The elegant and consistent syntax of Ruby makes code easy to understand and write while still offering advanced meta programming features that can be taken advantage of to solve complex problems. [31] It has been commended as a language that stays out of the way, allowing the programmer to focus on solving the task at hand instead of dealing with language related issues. [63]

Ruby has multiple different implementations. This thesis will consider CRuby which is the notation for the official Ruby implementation developed by Yukihiro Matsumoto [31], Rubinius which is an alternative implementation of Ruby that allows programmers to take better advantage of multi core processors with concurrent programs [25] and JRuby which is an implementation that executes Ruby code inside the Java virtual machine. [10]

2.1.1 The interpreter and compiler

Ruby was originally a fully interpreted programming language, where the interpreter first tokenizes and parses the text written in the ruby file, the parsed text was used to create nodes to represent the code written by the programmer and these nodes were saved in an abstract syntax tree (AST). Ruby would then run the code by walking the AST executing the nodes one by one. [58] The biggest problem with this approach to program execution is that it becomes very slow compared to compiled languages. This lead to early Ruby versions suffering from severe performance problems.

Koichi Sasada showed that the speed of Ruby programs can be significantly improved by adding another step before execution, namely compiling the code into instructions that can be executed in a virtual machine. With the creation of Yet Another Ruby VM (YARV) Sasada was able to optimize Ruby execution and obtain a speed up of multiple times faster execution compared to the original Ruby interpreter. [56] In fact the YARV virtual machine was so successful that it was added to Ruby as the default execution environment for Ruby code following release 1.9. Pat Shaughnessy showed that Ruby 1.9 can be over four times faster than Ruby 1.8 in executing simple programs [58].

2.1.2 Garbage collection

Garbage collection (GC) has a huge impact on application performance but is very hard to program into a language. Many algorithms have been developed to try to make garbage collection as effective as possible, but they have yet to reach the speed of manual memory management languages like C. Ruby comes with automatic garbage collection based on the mark-and-sweep algorithm invented by John McCarthy in 1960. [50] In Ruby the garbage collector has three main responsibilities, allocating memory for new objects, identifying unused objects and reclaiming memory from unused objects.

The initial implementation of the garbage collection uses the relatively simple mark-and-sweep algorithm. It maintains a list of available entities in a free list, when Ruby needs memory for a new object it gets assigned one entity from the list. Once the list is empty the execution of the application is paused while the garbage collector traverses all allocated objects and marks any object that is still referenced in the application. Any object not marked after the traversal is swept by the algorithm and returned to the free list. Version 1.9.3 of Ruby introduced a feature called lazy sweeping, with this feature the garbage collector only sweeps a set amount of objects back to the free list. This reduces the time spent with execution paused each time the collector is run, but instead increases the frequency of which it has to run. Version 2.1 introduced a generational garbage collection algorithm where it keeps track of which objects are young and which are mature. Mature objects are more likely to continue living in the application therefore they do not have to be swept as often as the young ones. Therefore the garbage collector saves

time by not checking the mature objects in every garbage collection sweep. [58]

The fact that the execution of the entire Ruby application is stopped when garbage collection is run means that memory allocation has a large impact on application performance in Ruby. Allocate an unnecessary amount of memory and the garbage collector has to run more often leading to more pauses in execution and lower performance.

Because Ruby offers the programmers so much freedom in how problems can be solved with the language, knowing exactly how the high level functionality is implemented and what effects it has on the performance is essential to writing high performing Ruby applications. Alexander Dymo showed that commonly used iterator functions in Ruby can create up to three additional objects each iteration. [42] This means more objects that need to be handled by the Ruby garbage collector and slower run time of the application. Ruby offers much convenient functionality to help the programmer solve problems as easy and efficiently as possible, but using some of this functionality without understanding of the underlying implementation can lead to significantly slower applications and should be avoided when developing anything time critical.

2.1.3 Concurrency and global interpreter lock

With modern central processing units (CPU) having an increasing amount of cores instead of higher clock frequency we can no longer assume that the execution speed of our applications will increase as the processing power of our computers increases. In order to take advantage of the increase in power we need to write code that takes advantage of multiple cores. Starting with version 1.9 Ruby offers real operative system level threading rather than green threads that are handled solely by the interpreter [63]. By supporting threads Ruby offers concurrent execution of applications, however it does not support parallel execution of Ruby code, meaning that in this implementation of Ruby multiple sections of Ruby code will never execute at the same time. The reason for this restriction is the use of something called global interpreter lock (GIL). GIL is a mutual exclusion (mutex) lock that threads need to acquire before they are allowed to execute code in the interpreter. Only one thread can hold the lock at any time so additional threads have to wait their turn. The developers of Ruby have implemented the GIL in order to reduce the risk race conditions in both the internal Ruby code as well as the application code, but this puts a limit on parallelization of Ruby applications. There is one important exception to the limitations of the GIL and that is blocking input/output (I/O) operations. When one thread gets blocked waiting for a heavy I/O operation the GIL lock is released and another thread can acquire it to start execution. This allows for I/O heavy applications to receive a speedup using multiple threads even though the GIL prevents parallel execution. [61]

2.2 JRuby

JRuby is Ruby running inside the Java virtual machine (JVM). Using Java to power Ruby comes with many advantages, applications using JRuby get access to real multi threading in the JVM without the many limitations of the GIL. JRuby also allows for seamless integration between Ruby and Java code giving developers access to the large library of tools available within the Java ecosystem as well as the option to write time critical parts of the application in Java instead of Ruby. By writing parts of the application in Java they can take advantage of the speed of Java while still benefiting from the fast development of Ruby.

JRuby supports multiple ways of executing Ruby code. The simplest way works much like in CRuby 1.8 by reading the text of the program file, interpreting the instructions and executing the code. The biggest disadvantage with this way of executing code is that much like in the case of CRuby the performance of the applications gets degraded. For better performance JRuby comes with a just in time (JIT) compiler. The JRuby JIT compiler analyzes the application at run-time and finds the parts of the application where the most of the execution time is spent. The compiler uses the live information from run-time in order to optimize and compile these sections into Java native byte code that the JVM can execute straight away. The JIT compiler makes the application run faster over time as the compiler gathers more information about the application. It can also make the start-up time of the application much faster as there is no need to compile the entire project at start-up. JRuby also supports ahead of time (AOT) compilation where the entire code base is compiled into Java byte code before run-time, for most applications JIT is preferable due to the compiler having access to run-time information for optimizations when compiling but AOT compilation is a viable option on platforms where JIT not available. [52]

JRuby has access to the very fine tuned garbage collector built into the JVM. The GC works by copying the objects still in use to another segment of memory then reclaiming all the memory of the first segment. One of the biggest advantages of the JVM GC is that it allows for concurrent garbage collection, meaning that the application can continue executing even as memory is being reclaimed, this offers a big advantage over the garbage collector of CRuby where the application execution has to pause during collections. It also supports generational garbage collection where mature objects which are more likely to survive the collection are separated from the young ones so the collection of mature objects can run less frequently. [58]

JRuby uses the threading functionality of the Java JVM and does not enforce any GIL to inhibit real parallel threading in applications. Instead of a GIL to protect the internal functionality JRuby uses fine grain locking for increased parallelizability. JRuby however, does not implement any protection from race conditions in the code it executes, therefore the responsibility for correct management of multi threaded applications falls on the developers. Due to the fact that code is executed in the JVM, JRuby does not support extensions relying on the third party C API available to programmers in CRuby. Third party extensions have to be rewritten in Java for

compatibility with JRuby. [61]

2.3 Rubinius

Rubinius is an alternative implementation of Ruby that puts emphasis on performance and concurrency with its support for parallel execution of threads. [25] Rubinius offers a very unique implementation of Ruby with large parts of the Rubinius kernel written in pure Ruby code. This gives developers the option to easily look up implementation details about built in Ruby classes and methods. It uses a virtual machine implemented in C++ that much like JRuby supports JIT compilation into byte code that can be executed at a significantly faster speed than purely interpreted code.

Rubinius uses much like JRuby an advanced copying garbage collection algorithm that supports generational and concurrent collection. The garbage collection algorithm in Rubinius is based on an algorithm called Immix. [58] Immix uses a mixture of marking and copying to achieve better performance, it has been shown to increase total application performance of 7% to 25% compared to many other algorithms. It introduces opportunistic copying to a marking garbage collection algorithm making it possible to rearrange the objects in memory for maximal collection and allocation efficiency. [39]

Rubinius offers real system level threading without a GIL and uses fine grain locking to protect internal operations from race conditions. Unlike like JRuby Rubinius does support the C extension API provided by CRuby, trusting that the extension developers will make them capable of running in a parallel execution environment. [61]

2.4 Ruby on Rails

Ruby on Rails (RoR) is an open source web framework with emphasis on sustainable productivity. [20] RoR makes development, deployment and maintenance easier and it is one of the biggest reasons to Ruby's rise in popularity for web development. It is an Model-View-Controller (MVC) framework following two core philosophical principals. Don't repeat yourself (DRY) which means that every piece of knowledge in a system should reside in only one place, thus removing any duplicity in configuration. The other core principle is convention over configuration (COC). RoR is a very opinionated framework that comes with default settings that fit most projects. The COC principle means that a developer should only have to configure settings that go against the RoR conventions. Agile development is one of the foundations of RoR, the framework makes it easy to deliver early working prototypes and adapt the project to changing requirements. [55]

The expressiveness of the Ruby language combined with the power of RoR makes development faster than with many other languages and frameworks. A Ruby on Rails developers can get a working prototype up and running quickly which is very

important in todays world of rapid prototyping and agile development. The framework delivers a complete stack for everything from connecting to a database, rendering views to providing RESTful services that other applications can consume. [45]

In addition to the original CRuby Ruby on Rails is supported by JRuby [52] and Rubinius [26]. The performance of Ruby on Rails can differ depending on the interpreter. [52] states that Ruby on Rails applications can perform better by just switching from CRuby to JRuby and taking advantage of the advanced multi threading and garbage collection of the JVM.

2.5 Caching

Caching is very important for performance in computer applications. The central processing unit (CPU) of a modern computer has multiple cache memories. These memories are small buffer memories which are very fast. The CPU uses them to store information which is believed to be in use so that when it needs some data next time it may be able to find it in the cache instead of having to request it from the slow main memory. [59]

Caching can also be used inside applications to buffer information which is likely to be needed again in the memory of the computer. By caching information in memory the application can avoid repeated expensive fetching operations to get information from the hard drive or external services. Caching can be done in different layers from caching the results of an expensive application operations to caching the entire response to a request and serving subsequent requests with the exact same information. [24]

2.6 Scalability

Scalability is a common term when describing multiprocessor systems and distributed systems. It is a term that can be used to describe many different attributes of a system. [47] tries to find a useful and rigorous definition for the term but fails.

The term scalability is sometimes used to describe the efficiency of a system in a multiprocessor environment. This is the definition which is used in this thesis project. It is used as a metric for how the application is able to take advantage of multiple cores in a modern computer system.

2.7 Databases

When storing large amounts of information that needs to be easily accessible it is most likely a good idea to use a database of some kind. In 1970 [40] presented a new way of storing data, the ideas from the paper published by Codd came to evolve into the relational database systems that are used in many modern system. Relational databases offer structured query language (SQL) which is an advanced high-level language for accessing information stored in the database.

At the foundation of relational databases lies the transaction properties abbreviated as ACID. The A stands for Atomicity meaning that all transactions are all-or-nothing, that is they are either fully committed to the database or rolled back. The C stands for consistency which basically means that transactions can not put data elements in a state that breaks any constraints set on those elements. I stands for isolation an has the effect that each transaction should appear to be executed as if there are no other transactions. Lastly the D stands for durability and states that a completed transaction must never be lost even in the event of system failures. Modern relational databases offers an easy to use yet very powerful way of querying for data while maintaining high durability even in the event of failures. [44]

With the rising need to store large amounts of unstructured or semi structured data a another type of database has increased in popularity, the NoSQL databases offer a fast and efficient way of storing large amounts of data that may not be suitable for a relational database. By sacrificing the ACID properties the NoSQL databases are in some cases able to achieve higher performance than relational databases.

NoSQL databases are generally divided into three categories, key-value store which is an index for storing data accessible by a specific key, column-oriented database which stores information in columns containing related data. The last category of NoSQL is document-based stores which orders the data into collections of documents instead of tables. The structure of individual documents can be different from others in the same collection, this gives more freedom to how the unstructured data is represented. [48]

But NoSQL is not only popular with large applications that process mainly unstructured data, [54] showed that with modest-sized structured data NoSQL can offer performance gains over relational databases. Combine performance and flexibility of information structure with a querying syntax that is easy to understand and does not require the developer to learn a new language just to query the database for information and it becomes apparent why NoSQL is gaining popularity. [48]

MongoDB [14] is one popular NoSQL implementation, it offers a document oriented database system that represents documents in a form similar to the simple JavaScript object notation (JSON) [11] making it easy for developers to visualize objects as stored documents. Documents are grouped into collections and can be accessed using an easy to understand querying language. MongoDB offers great performance and simple horizontal scaling, so that additional servers easily can be added to increase capacity. [14]

2.8 Related work

Much work has been done in the area of optimizing program execution, but since this project focuses on the optimization of the high level programming language Ruby and the web framework Ruby on Rails many low level optimization techniques are not viable. This chapter will focus on related works that are relevant to optimizing

Ruby and Ruby on Rails for both better performance and general scalability.

2.8.1 Ruby performance and scalability

In a scripting language like Ruby the speed of the interpreter has a huge impact on application performance. Yet Another Ruby VM (YARV) [56] showed how the performance of Ruby can be significantly increased with a Ruby virtual machine that compiles the code before running it. YARV performes simple algorithms many times faster than the fully interpreted Ruby version. With version 1.9 of Ruby YARV was included as the default virtual machine for Ruby.

Another big bottleneck in Ruby performance is the global interpreter lock (GIL) which inhibits parallel execution of Ruby code. [53] demonstrated how the GIL can be eliminated with the use of hardware transactional memory to achieve a 1.2-fold speedup with Ruby on Rails. Hardware transactional memory comes with many restrictions as to how much memory can be used in one transaction. It requires that support is implemented on many different platforms before it becomes a viable option for replacing the GIL. [51] looks at different options for eliminating the GIL in the dynamic programming language Python and suggests that software transactional memory may be a viable option. Software transactional memory offers arbitrarily long transactions and does not require support from the hardware but comes with large performance loss compared to hardware transactional memory. More research is needed into both hardware and software transactional memory before any of them can offer a viable replacement for the GIL. This project differs from this research in how it puts focus on higher level optimizations rather than changing language implementation specific details in order to increase overall performance.

Raw performance is not the only factor when trying to build fast web applications, scalability is another important factor. Being able to utilize available sever resources is important as well as the ability to distribute functionality over many servers [57] looks at how DRuby and Rinda can be used to build Ruby applications distributed over many servers with remote method invocation (RMI) and parallel coordination. DRuby offers an approach to distributed computing that is very similar to regular Ruby applications, thus shortening the learning curve for Ruby developers. By showing concepts, design as well as the implementation of DRuby the paper seeks to demonstrate that Ruby and DRuby is a viable and robust infrastructure for building distributed applications.

Due to the GIL inhibiting parallel execution of Ruby code it is the general convention to use multiple processes instead of threads to achieve better performance and utilize multi core processors. However, when introducing multiple independent processes executing the same code it gets harder to debug the software [38] shows how fork handlers can be used for a better debugging experience of multi process applications with the open source debugger Dionea.

[43] discusses the implementation of a JIT compiler for the dynamic programming language JavaScript. It identifies hot spots in program execution and records

the traces of the execution path together with the types of variables in the code. Since JavaScript is a dynamic language the type of a variable is determined dynamically at run time and may differ in different executions of the same section of code. The recorded hot spots can be used instead of regular code interpretation next time that section of code is to be executed if the variable types coincide. The recorded traces can be executed with much higher speed than interpreted code, thus providing a significant speedup in hot spot heavy code. The concepts discussed can be applied to Ruby as well since it faces the same issues due to dynamic variable types. [65] discusses the implementation of a trace based JIT compiler for dynamical programming languages using a hierarchical structure of virtual machines. By running a virtual machine executing the code of the dynamic language inside another virtual machine that implements trace based JIT compilation and optimization, the authors make it possible to use one well optimized JIT compiler for multiple different dynamic languages. This can be done by just implementing the inner VM that executes the code and exposes the information needed for the outer VM to perform the JIT compilation. The big advantage with this approach is that it would become much easier to implement JIT compilation for a dynamic programming language and the performance of a mature and well optimized virtual machine can be taken advantage of to increase the performance of programming languages like Ruby.

One advantage with dynamic programming languages is that the variable types are determined at run time instead of during compile time like with static type languages. Dynamic variable types allow a variable to be an integer one time and a string the next time the same piece of code is executed. With dynamically typed languages more freedom is given to the programmer to manipulate the variables but this comes with a performance cost. Since a variable can have any type at any time, the interpreter or virtual machine executing the code must continuously lookup the variable types and this type checking is responsible for a significant part of program execution time. [64] suggests a dynamic intermediate representation of code for scripting languages, by encoding variable types at each point of execution the overhead created by costly type checking is reduced. Implementing this technique for the Lua scripting language showed performance speed-ups of 1.3x on average in general benchmarks.

2.8.2 Database performance

Databases is a popular field of research and there are many works describing low level optimizations to increase the performance of different database systems. [49] compares the performance of multiple NoSQL implementations to SQL for different common database tasks. They show that performance of NoSQL varies greatly and while some of the implementations perform overall better than SQL, that is not the case for all NoSQL databases.

[54] takes another approach and compares the performance of a popular NoSQL implementation called MongoDB with SQL Server using a modest-sized structured database. With this comparison they show that MongoDB can perform equally well

or better than the SQL server in test cases not dealing with aggregate functions. This shows that NoSQL can be utilized for structured as well as unstructured data, making it a viable database back end for web applications even if the data is rather structured. [60] on the other hand argues that the performance gains achieved in NoSQL are due to the logging, locking, latching and buffer management that SQL databases offer for a more reliable and an durable system. The author claims that by eliminating these SQL bottlenecks performance on pair with NoSQL can be achieved.

Chapter 3

Methodology

Previous chapter provides some background information about Ruby on Rails as well as a presentation of other works that are related to this project. In this chapter we present some more in depth information about the tools and methods that were used for optimization.

A computer is an advanced system of components working together to perform complex tasks and calculations. The computer architecture has evolved and become much more sophisticated since the invention of the first computer systems. In early computers memory and processor performance were on the same level, but as the technology advanced a gap between the performance of the CPU and the main memory arose. Over time this gap increased to the point where the processor has to wait for a long time to fetch data from the memory.

In order to combat the increasing performance gap between the CPU and main memory intermediate levels of memory were introduced in the form of caches. A modern CPU often has three levels of caches with different speed and capacity to avoid having to fetch information from the slow main memory if possible.

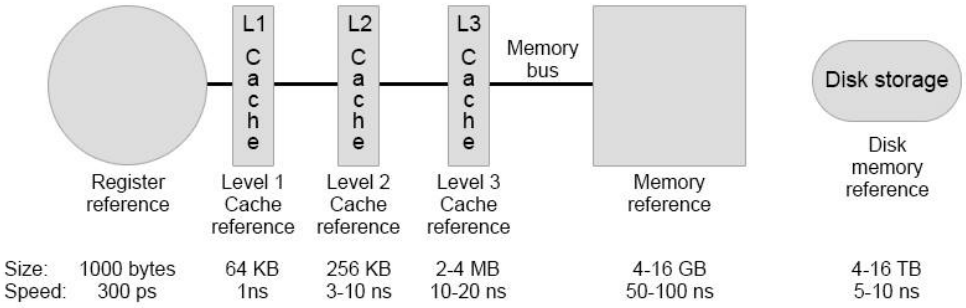

Figure 3.1. The memory architecture of a modern computer. Adapted from [46]

Figure 3.1 shows the memory architecture of a modern server computer. It

contains a small set of very fast registers which can be used for storing and retrieving a small amount of data, the registers are very fast but also very expensive thus the very limited size. In addition to the registers three levels of caches are used to improve performance, the L1 cache is the fastest cache but the smallest, the L2 cache can fit more information but is substantially slower. The L3 cache is the largest and slowest cache, however it is still much faster than the main memory. [46]

One very important characteristic of program execution is that data that has just been used and its neighboring data are likely to be needed again in the future, this is called program locality. Program locality is divided into two categories, temporal locality which is the concept that a resource that is referenced at one point is likely to be referenced again some time in the future, spatial locality on the other hand is the concept that resources near a referenced resource are likely to be referenced in the future. [13]

CPU caching has significant impact on application performance. In order to create high performing applications it is important to take advantage of the caching to achieve better performance. However, there are many more factors to application performance than utilizing the cache in the CPU. In this chapter execution time, memory and garbage collection will be considered. How they can be measured, analyzed and optimized will be the main focus for the rest of the chapter.

3.1 Execution time

Program locality has a big impact on execution speed, if an application has good locality that means that the processor has to spend little time waiting to fetch resources from the main memory, but on the other hand if the application has bad locality, not only does the processor have to wait to fetch resources from memory, additional time is also wasted fetching spatially nearby located resources that will not be of any use. Many optimizations for locality is made by the compiler at compile time but it is also important for the developer to take these concepts in to consideration when writing high performance applications.

In order to make an application run faster it is important to know where most execution time is spent because this is what needs to be optimized. A profiler can be used to help find the most CPU and memory intensive sections of an application.

3.1.1 Ruby-prof

Ruby-prof is a popular profiling tool for Ruby and Ruby on Rails applications. It is a fast profiling tool for Ruby code written as a Ruby C extension. [27] The primary use case for ruby-prof is to profile parts of a Ruby application to figure out where the largest performance bottlenecks are located. However, it can also be added as a RoR middleware to give an overview of the application performance, including the functionality that is handled by the Rails framework. Profiling results can be exported into multiple formats including the callgrind format used by the popular C and C++ profiler Valgrind [42]. This allows for the results to be visualized with

tools developed for Valgrind like KCachegrind [12]. Figure 3.2 shows a call graph visualization of the results from a CPU profiling using KCachegrind. Ruby-prof is

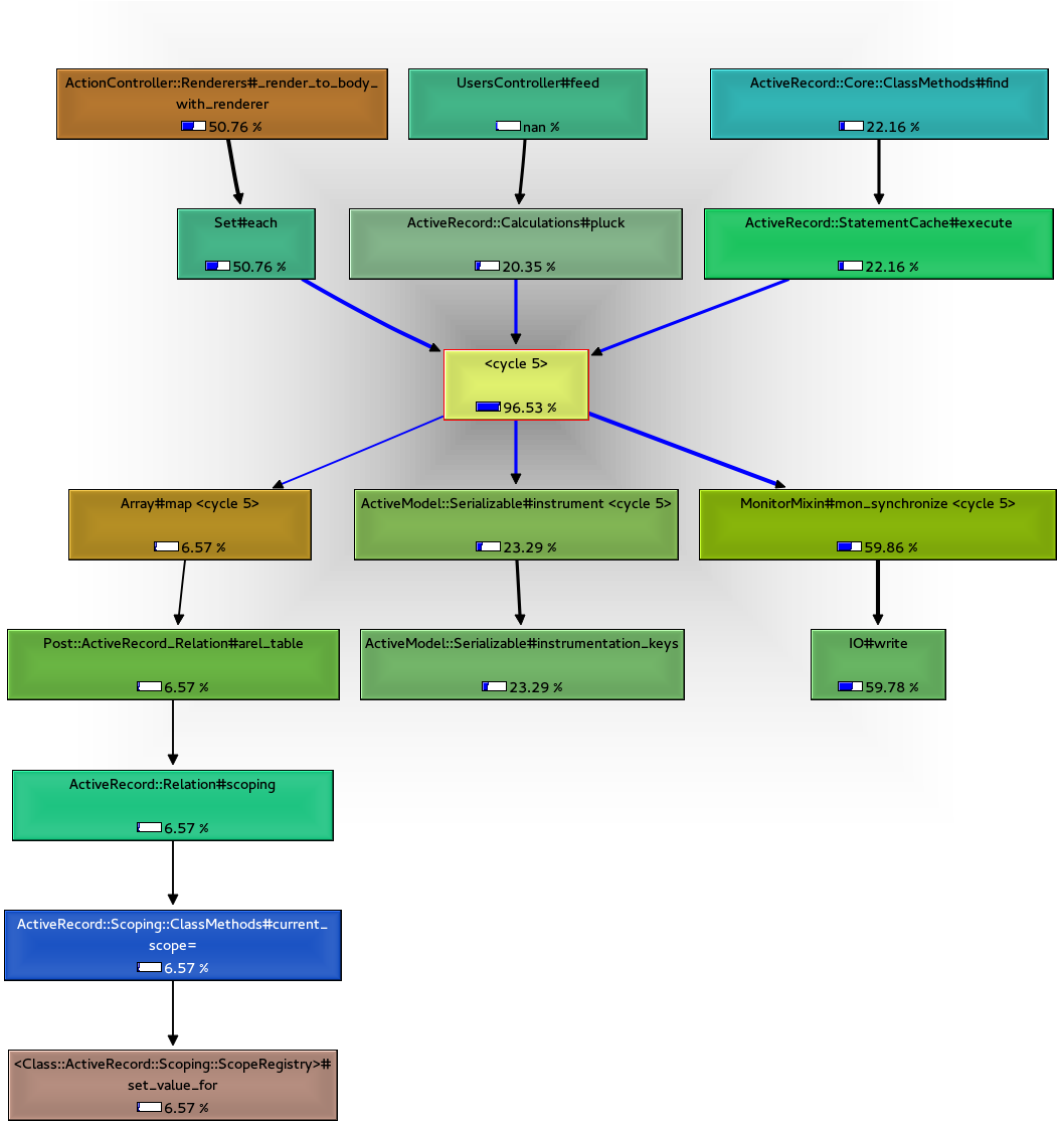

Figure 3.2. Call graph from CPU profiling of the feed endpoint in KCachgrind

mainly used for CPU profiling of Ruby applications but with a patched version of Ruby it is possible to obtain readings of either the size of memory allocated or the number of allocations made within a block of code. This makes it possible to use ruby-prof for memory profiling as well. [42]

3.1.2 JRuby profiling

One big advantage with using JRuby instead of CRuby is the ability to use tools from the very large Java ecosystem. Profilers are no exception to this, there are many tools available for profiling Java applications. One popular Java profiler is JProfiler, it offers a complete profiling suite with support for memory, CPU and thread based profiling. [9] An alternative to JProfiler is the YourKit Java profiler. It offers a similar feature set to JProfiler with CPU, memory and thread based profiling. [37]

3.1.3 Measuring execution time

Application execution time may not be the best metric for measuring the efficiency of a web application due to the fact that there are many factors that determine the user perceived application performance than merely execution time. It can however, it can give great insight into where calculation intensive bottlenecks of the application are located.

Measuring code execution time

After profiling an application and figuring our which parts of the code are the biggest performance bottlenecks the code can be optimized in order to increase performance. But too many factors affect the execution time of an application for it to be a viable measurement to determine how a small change to the code base affects the performance.

Listing 3.1. Benchmarking code in the assert_performance gem

```
def self.benchmark_code(name, &block)
    operation_results = nil
    read, write = IO.pipe
    (0..30).each do |i|
      # Force GC to reclaim all memory used in previous run
      GC.start

      pid = fork do
        # GC extra memory that fork allocated
        GC.start
        # Disable GC if option set
        GC.disable if ENV["RUBY_DISABLE_GC"]

        # Store results in a between runs
        benchmark_results = File.open("benchmark_results_#{name}",
        "a")
        elapsed_time, memory_after, memory_before = nil
        begin
```

```
            ActiveRecord::Base.transaction do
              memory_before = `ps -o rss= -p #{Process.pid}`.to_i
              elapsed_time = Benchmark::realtime do
                operation_results = yield
              end
              memory_after = `ps -o rss= -p #{Process.pid}`.to_i
              raise PerformanceTestTransactionError
            end
          rescue PerformanceTestTransactionError
            # Rollback database
          end
          # Skip first run to exclude cold start measurements
          if i > 0
            # Store runtime
            benchmark_results.puts elapsed_time.round(6)
          end
          benchmark_results.close
          GC.enable if ENV["RUBY_DISABLE_GC"]

          read.close
          results = {results: operation_results,
                     memory: (memory_after - memory_before)}
          Marshal.dump(results, write)
        end
        Process::waitpid pid
      end
      measurements = File.readlines("benchmark_results_#{name}").
      map do |value|
        value.to_f
      end
      File.delete("benchmark_results_#{name}")

      average = average(measurements).round(5)
      stddev = standard_deviation(measurements).round(5)

      ...
    end
```

To be able to measure the performance of specific sections of code the ruby gem `assert-performance` was implemented from specifications in [42]. Listing 3.1 shows a selection of the code for measuring execution time. The gem can run an arbitrary block of Ruby code and measure the time taken for execution. In order to minimize the effects from external factors the code is executed 30 times and the mean value is calculated as well as the standard deviation.

This gem is used for measuring the impact of optimizations on specific parts of Ruby on Rails code. By measuring the execution time before and after a specific optimization the results can be compared. This can be used as a basis for deciding the efficiency of the optimization. Measurements made with the gem are automatically persisted to Parse [18] for further evaluation at a later stage.

Measuring database queries

Database performance has a huge impact on the overall performance in database heavy applications and applications developed with Ruby on Rails often fall into this category. Active Record [1] offers an abstraction from the underlying database layer. This allows developers to write code that is independent of the database implementation. Queries in Ruby format are automatically converted into queries in the query language supported by the database. This makes developing applications easier since the developers do not have to worry about database implementation specific details. However, with this convenience some control is lost, changes to the code can lead to unanticipated changes to the queries generated by Active Record. These query changes may negatively affect the performance of the application. In order to monitor changes in queries the `assert-performance` gem implements functionality to catch all database queries executed for a specific block of code. Using Active Record provided event hooks the generated queries are logged and persisted to Parse. This makes it easier to understand how code optimization affects queries to the database and allows for optimization without writing database dependent queries by hand.

New Relic RPM [15] is a performance monitoring and profiling Ruby gem which can be used in a production environment to monitor vital server stats and application performance. In addition to general performance statistics the RPM gem offers detailed database performance data. The most time consuming queries can be analyzed and the gem offers general suggestions for how to fix performance issues. Profiling in a production environment allows for identifying bottlenecks that may not be discovered when profiling in the development environment and RPM makes that possible without adding significant performance overhead.

3.2 Memory

Memory is a finite resource in a modern computer system. Memory allocated by one application has to be freed at some point in time so that the computer does not run out of resources. Some programming languages like C have manual memory management where it is the developers responsibility to return any allocated resources back to the system after usage. [41] This gives full control over the application memory to the developer but this power can be a double edged sword. On one hand it makes it possible to write better performing applications by optimizing memory management. On the other hand it opens up the possibility for memory

mismanagement which can cause memory leaks or other severe malfunctions in the application.

Unlike C Ruby has fully managed memory, relieving the developer of the responsibility of freeing allocated memory. This allows all focus to be put on developing application functionality. This convenience comes with a hefty performance price however, in CRuby the entire execution pauses so that the garbage collection algorithm can free unused objects. If this pause in execution happens at a crucial time in performance critical applications it can cause severe problems. It is therefore essential to not allocate unnecessary objects in order to minimize the amount of work the garbage collector has to perform.

Much like the execution time, memory usage can also be profiled using similar profiling and measurement tools.

3.2.1 Stackprof

Stackprof [34] is a memory profiler that provides a different approach to profiling memory. Instead of measuring the amount of memory allocated in different parts of the application it focuses on the number of objects allocated [42]. Stackprof can help locate where unnecessary objects are created in the application.

3.2.2 Measuring memory usage

Profiling an application can give great insights into which parts consume the most memory, but the act of profiling changes the environment of the application compared to how it is used in production. What matters the most when measuring the results of an application optimization is not the improvements to isolated parts of the application but the overall improvement. Exact measurements of memory usage can be difficult since multiple processes may be used for parallelism and scaling.

To make the measurements process more manageable the metric used to evaluate the applications memory consumption is the total memory consumption of the entire system. Any change to the applications memory consumption is directly reflected in the total system consumption. Nmon [16] is a performance monitoring tool for Linux systems that give access to various performance related data like CPU, memory, network and disk usage. Using NMOMVisualizer [17] data exported from Nmon can be visualized with different graphical representations.

3.3 Response time

One of the most important metrics for web applications is the response time. Users only care about how responsive the application feels when they use it, not how memory efficient or well optimized it is. Given that a well optimized and memory efficient application will often lead to lower response time, but this is not always true. There may be factors outside the application that affect response time negatively. In order to get measurements that as accurately as possible reflect the experienced speed

of an application response time is used as one of the main metrics for measuring efficiency of application optimizations.

3.3.1 Measuring response time

To eliminate any network related interference all measurements of response time is done locally on the same virtual server instance as the application server is running on, that way external factors like network congestion has no effect on the results. Measuring response time under perfect conditions where there is only one concurrent user has very little correlation to real world conditions where multiple users are requesting information from the server at the same time. With this in mind measurements of application response time are done while the application server is under simulated load. This gives insight into how the application performs under pressure.

Apache Bench

Apache Bench [3] is a very simple web server benchmarking tool that is pre-installed on many Linux and Unix operating systems. It provides a simple command line interface for making a set of concurrent HTTP request to a specific address and offers a very simple overview of the results of the performance test, including mean request time, requests per second and transfer rate.

One of the biggest advantages of using Apache Bench is the simplicity, it requires no additional software to be installed on many server operating systems and it is very easy to use. It can give a good overview of application performance but may not offer enough information for any deeper analysis.

Httpress

Httpress [6] is another light weight HTTP server benchmarking tool very similar to Apache Bench, but with the added feature of multi-threaded testing. With httpress it is possible to utilize multiple threads when stress testing a server, which can be very beneficial when very high intensity tests are required.

JMeter

Apache JMeter [8] is a mature and well used open source load testing software written completely in Java. It offers a more extensive testing suite than both Apache Bench and Httpress with functionality to create advanced test cases simulating users doing real tasks. Results from JMeter can be analyzed and visualized in different ways. It offers multi threaded benchmarking allowing high performance load testing utilizing multiple CPU cores.

With the Ruby gem RubyJmeter [30] test plans for JMeter can be defined with Ruby code which is automatically transformed into correct configuration files for JMeter. Using Ruby to create test plans makes it easier to create more advanced

and dynamic test plans. Current database table ids can be fetched and used in the test plan without having to manually rewrite the configuration file.

3.4 Garbage collection

Garbage collection can have huge impact on application performance but it is a very hard problem to solve. There are many theories and strategies for how to make garbage collection as efficient as possible and minimize its impact on the performance of the application. Garbage collectors for programming languages often have very optimized default settings to provide good performance for most applications. Performance gains can be achieved by optimizing the settings to the requirements of a specific application. CRuby, Rubinius and JRuby all have different approaches to garbage collection, each of which has specific advantages and disadvantages.

3.4.1 Measuring the garbage collector performance

The time spent on garbage collection has a significant effect on overall application performance, especially when the garbage collection pauses application execution as in CRuby. Ruby offers access to various garbage collection related statistics and profiling data that can be used to evaluate and improve the performance of the garbage collector. [42] This information can be obtained in a similar way for both CRuby and Rubinius.

The JRuby implementation of Ruby executes in the Java VM and therefore supports tools written to analyze garbage collection in Java applications. Since Java is a popular programming language in time critical applications much effort has been put on optimizing the JVM garbage collector to make it have as little negative impact on application performance as possible.

3.5 Benchmark environment

In a modern computer there are almost always multiple applications running at the same time. Some of these applicaitons may be part of the operating system, whilst others are run by the users. At any given time there are most likely more processes running than available cores on the computer so the operating system has to switch between the applications, allowing each application a fair share of time to execute code on the CPU. There are many different algorithms for how to schedule application execution but one thing they have in common is that execution is switched between processes to distribute execution time on the CPU [62]. The interleaving of process execution form an non-deterministic pattern making it impossible to predict the sequences of execution. Normally this has very little effect on application execution. For time critical applications however, process scheduling may cause severe problems. The operating system performing a context switch to

allow some other process to execute at the wrong time can have a negative effect on application performance.

To minimize external interference when benchmarking the application all tests are performed in an isolated environment in the form of an Amazon AWS virtual server. [2] Each instance is provisioned with the same applications using the application provision tool Chef [4] and the database is imported using a SQL dump created from the database used for development and testing. Amazon AWS C4.Xlarge instances with a four core CPU and 7.5 GB ram are used to be able to see how the application scales with CPU utilization and memory usage. Server computational performance is nowadays mostly scaled by increasing the number of cores available to the system, in contrast to increasing the clock frequency. Therefore it is important to consider how well the optimized application can utilize a multi core CPU.

Chapter 4

Design and implementation

In order to be able to evaluate different optimization techniques and use the tools presented in earlier chapters an application is required. For that reason a Ruby on Rails application was developed, it was implemented using specifications received from Slagkryssaren AB. [33] A summary of the received specifications can viewed in table 4.1. To focus on optimizing server side operations the application was implemented as an API with no front end. Drawing from experiences at Slagkryssaren on what type of applications commonly suffer from scaling issues the application was modeled after the very popular image sharing social network Instagram.[7]

The application is a REST style API providing the back-end functionality of a simple image sharing social network. Users can create an account by submitting a sign up request to the correct API endpoint. In order to access most of the services users need to login with their account details and receive a request token that has to be included with requests that require authentication. Logged in users can post pictures, interact with other users by viewing and commenting their pictures. Users can follow other users and each user has a personal feed that displays their own posts as well as any posts made by the users they are currently following.

Table 4.1: Specification of evaluation API

Path	**Params**	**Response**
user/signup	String: username, String: email, String: password, Date: birthdate, String: description, String: gender,	201, bool:success
user/login	String: username, String: password	200, bool:success

Continued on next page

Table 4.1 – *Continued from previous page*

Path	Params	Response
user/:id		200, bool: success, object<User>, int: following, int: followers
users	Integer: offset, Integer: limit	200, bool:success, array<User>
user/:id/followers	integer: offset, integer: limit	200, bool:success, array<User>
user/:id/following	Integer: offset, Integer: limit	200, bool:success, array<User>
user/:id/feed	Integer: offset, Integer: limit	200, bool:success, array<Post>
post/create	File: image, String: description, Array: tags, Array: user_tags	201, bool:success
post/:id/update	String: description, Array: tags, Array: user_tags	200, bool:success
post/:id		200, bool:successobject<Post>
post/:id/like		201, bool:success
post/:id/likes		200, bool:success
post/:id/comments	Integer: offset, Integer: limit	
comment/create	Integer: post_id, String: comment, Array: tags, Array: user_tags,	201, bool:success

Continued on next page

Table 4.1 – *Continued from previous page*

Path	Params	Response
comment/:id/update	String: comment, Array: tags, Array: user_tags	200, bool:success
comment/:id		200, bool:success, object<Comment>

The feed helps users stay up to date with pictures shared by friends and family and is a core feature of the image sharing social network. A user's feed may contain many posts. In order to keep the response at a manageable size and decrease response time the application uses pagination to allow users to step trough a large amount of posts divided into multiple pages.

Listing 4.1. Personalized feed

```
def feed
  user = User.find(params.require(:id))
  offset, limit = pagination_values
  feed_users_ids = user.followings.pluck(:id)
  feed_users_ids << user.id
  feed_posts = Post.where(author: feed_users_ids)
               .order(created_at: :desc)
               .offset(offset).limit(limit)
  response = {
    success: true,
    offset: offset,
    limit: limit,
    result: feed_posts
  }
  render json: response, status: :ok
end
```

Listing 4.1 shows the unoptimized code for the user feed endpoint, it finds the ids of all the users the current user is following then fetches the posts made by the user himself and his followings. The results are sorted after time of creation and the correct pagination offset and limit is applied. The `feed` method shows how Ruby on Rails integrates tightly with the database using the Active Record ORM. Ruby models are automatically instantiated from the corresponding tables in the database. The fetched information can be used as any Ruby object with its own methods and attributes. This database abstraction makes the models independent of the underlying database, therefore the database system can be changed without having to change application code.

Listing 4.2. User model

```
class User < Active Record::Base

  ...

  has_many :user_tags
  has_many :posts, foreign_key: 'author_id'
  has_many :likes
  has_many :tagged_posts, through: :user_tags, class_name: 'Post',
           source: :post

  has_many :user_followings
  has_many :followings, through: :user_followings

  has_many :inverse_user_followings, class_name: 'UserFollowing',
           foreign_key: 'following_id'
  has_many :followers, through: :inverse_user_followings, source: :user

  ...

  def as_json(options = {})
    super({except: [:password_digest, :token], include: :posts,
           methods: [:following_count, :followers_count]}.merge!(options))
  end
end
```

Listing 4.2 shows a selection of the code for the User model, the user has many associations to other models which are resolved through Active Record. They can then be accessed directly from the user object. The relations between tables are maintained by Active Record independently of the underlying database system, thus allowing associations to be used even if the database system does not support relational enforcement using foreign keys. The followers association is a many to many relation which is automatically created using a third intermediate table, this makes it possible to access many to many associations from both sides of the association just like any regular one to many association.

The `as_json` method defines which attributes of the object are to be serialized when it is converted to JSON. Private fields like `password_digest` and `token` are excluded from the JSON object and the results from the `followers_count` and `followings_count` methods are included to show how may people the user is following and how many other users are following him. The association to posts is also included, making all posts by the user embedded inside the serialized JSON object.

Chapter 5

Optimization

This chapter presents an overview of the how Ruby on Rails optimization techniques and how performance was evaluated in this thesis project. A subset of the optimization techniques presented in this chapter were implemented and evaluated in this thesis project, the results from this evaluation is presented in the results chapter.

5.1 Evaluation endpoints

In order to manage the project workload a subset of endpoints were chosen as focus for optimizations, the same API endpoints were also used for performance benchmarking to measure the effects of the optimizations. The optimization and benchmarking endpoints were chosen on the merits of the importance of performance for user experience as well as computational complexity of the operation. The user is less likely to be bothered by additional waiting time for operations that are more rarely used like creating an account, post or comment than browsing posts, comments or their feed. Some API endpoints offer pagination, allowing the user to step through pages of results. Here performance is very important for the user experience as the user will make multiple requests to the application in one session and any additional response delay will quickly become an annoyance. It is also important to use endpoints that do not change the state of the database as this may affect the results of other benchmarks.

5.1.1 Feed endpoint

The feed is a core feature of the image sharing application, users can view images shared by their friends and acquaintances, it supports pagination to divide the results into pages that can be requested separately. Listing 4.1 shows the unoptimized code for the feed endpoint. It offers the highest code complexity as posts made by the user as well as any users he or she is following is gathered, organized and presented. The importance of the feed endpoint for application functionality as well

as its inherent code complexity makes it a good endpoint for measuring application performance.

5.1.2 Followers endpoint

The followers endpoint shows the ids and usernames of all people following a specific user. Because its specification requires that it shows only id and username of the users it adds additional complexity as the default way of serializing the user model in JSON has to be overridden in order to support this custom representation.

Listing 5.1. Followers endpoint

```
def followers
  offset, limit = pagination_values
  user = User.find(params.require(:id))
  response = {
    success: true,
    offset: offset,
    limit: limit,
    result: user.followers.offset(offset).limit(limit)
  }
  response[:result].define_singleton_method(:as_json,
  -> (args) { super(only: [:id, :username], include: [], methods: []) })
  render json: response , status: :ok
end
```

Listing 5.1 shows the unoptimized code for the followers endpoint, the complexity of overriding the object JSON representation can lead to performance issues, therefore it was chosen as one of the benchmarking endpoints.

5.1.3 Post comments endpoint

Post comments is a core feature to give users the ability to interact with other users and make the picture sharing a more social experience. Every post can have multiple comments and users may repeatedly check the comments of interesting posts, performance is therefore essential for a good user experience.

Listing 5.2. Post comments endpoint

```
def post_comments
  offset, limit = pagination_values
  response = {
    success: true,
    offset: offset,
    limit: limit,
    result: Comment.where(post_id: params.require(:id))
              .offset(offset).limit(limit)
```

```
  }
  render json: response, status: :ok
end
```

Listing 5.2 shows the unoptimized code for the post comments endpoint, it relies on the association between posts and comments in order to find all comments related to a specific post.

5.1.4 Users endpoint

The users endpoint lists information about all registered users, to make the listing of users more manageable the endpoint supports pagination to divide all the users into pages that can be requested separately.

Listing 5.3. Users endpoint

```
def index
  offset, limit = pagination_values
  users = User.order(:id).offset(offset).limit(limit)
  response = {success: true, result: users,
              offset: offset, limit: limit}
  render json: response, status: :ok
end
```

Listing 5.3 shows the unoptimized code for the users endpoint, it has little code complexity but it gives great insight into performance of simple information fetching and serializing of large sets of data to JSON. Many production applications mostly forward information from the database to the users without the need for complex program or database operations. Therefore it is important to consider this kind of simple operation in overall application performance.

5.2 Application optimizations

This section describes optimizations that can be performed on an application level in order to optimize performance.

5.2.1 Database queries

Most Ruby on Rails applications use the Active Record Object-relational mapping (ORM) [1] to easily manage the application models stored in the database. It is the default ORM for Ruby on Rails applications and offers an extensive and reliable interface for accessing data in the database as well as drivers for most popular database systems. With Active Record it is possible to query the database without having to write database queries specific to the underlying database system. Queries can be written using simple Ruby code which is converted into correct database queries. This is very convenient for developers as they don't have to write any

platform dependent query code, but this convenience comes with a performance cost. Active Record generated queries are not always efficient. To work as efficient as possible with the database it is important to only fetch the attributes that are needed for that functionality. Active Record however, does not make it convenient to query for a subset of columns for specific models [42]. This often leads to more data than necessary being fetched and more memory being allocated to store the model objects.

Listing 5.4. Unoptimized Active Record query for feed endpoint

```
user = User.find(params.require(:id))
offset, limit = pagination_values
feed_users_ids = user.followings.pluck(:id)
feed_users_ids << user.id
feed_posts = Post.where(author: feed_users_ids)
             .order(created_at: :desc)
             .offset(offset).limit(limit)
```

Listing 5.4 shows the unoptimized code for querying the database in the feed endpoint. It uses Ruby code to query the database with Active Record instead of writing manual queries. This is very convenient but not always efficient. Listing 5.5 shows an optimized version of the same query, here only the information required for the operation is fetched from the database instead of the entire model.

Listing 5.5. Optimized Active Record query for feed endpoint

```
feed_users_ids = UserFollowing.where(user_id: id)
                 .pluck(:following_id) << id
feed_posts = Post.select(:id, :description, :author_id,
             :created_at, :updated_at)
             .where(author: feed_users_ids).order(created_at: :desc)
             .offset(offset).limit(limit)
```

5.2.2 Counter cache columns

The user model should contain the number of users that the specific user is following and how many other users are following the user, in order to keep that information always up to date it is normally calculated by executing a count query for the models in the association. This can become a performance bottleneck when retrieving a large amount of models, since each individual model has to execute a count query.

To relieve this problem Active Record supports counter cache columns, which is an extra column added to the database that will contain the count for a specific association, this count is automatically incremented when new models are added to the association and decremented as they are removed. Counter cache columns eliminate the need for any unnecessary counting queries to the database.

5.2.3 Removing unnecessary middleware

Ruby on Rails is a very large web development framework with the goal to provide tools that cover all the most common use cases in modern web applications. This helps developers quickly develop applications for differing project requirements. This has the draw back that for any given project only a subset of all Ruby on Rails middleware (modules) are actually used, by disabling unused middleware the Ruby on Rails memory consumption can be decreased thus allowing more memory for other operations.

The `Rails::API` [21] is a gem that is specifically developed to provide a leaner Ruby on Rails specifically for API web applications. The main purpose of an API is to provide requesting clients with information, by removing middleware that is not needed by an API service the `Rails::API` gem makes it possible to lower the memory footprint of the application while still retaining all relevant Ruby on Rails features. Due to time constrains of this project `Rails::API` was not benchmarked separately but included in the optimized application suite, however [22] shows a 15% decrease in base memory consumption and 12% decrease in response time with `Rails::API` compared to using the full Ruby on Rails framework for a simple API web application. `Rails::API` provides such a good solution to making efficient API applications with Ruby on Rails that it will be included in the planned fifth release of Ruby on Rails. [23]

5.2.4 Caching

Ruby on Rails has built in support for caching reusable information in order to speed up the retrieval process. It is capable of caching everything from object models, database query responses to view rendering results and entire page responses.

SQL caching

SQL caching is a built in feature in Ruby on Rails which caches the result set from a database query for the entire request, if the same query is encountered again within the request the cached result set is returned without querying the database. After the request is finished the cached values are automatically destroyed and used resources are freed. [24]

Conditional GET caching

Conditional GET is part of the HTTP specification and provides a way for servers to tell clients whether the response has changed since the last request or not. When a browser makes a GET request to the server it passes `HTTP_IF_NONE_MATCH` and `HTTP_IF_MODIFIED_SINCE` headers to the server, this allows the server to determine if the information that is available in the browser's local cache is up to date or not. If the information is current then the server only needs to respond with a 304 HTTP header telling the browser to show the locally cached data, if the information on

the other hand has changed the request is served and the new fresh information is returned together with new header values for future requests. [24]

Listing 5.6. Conditional GET caching in the controller

```
class UsersController < ApplicationController
  def show
    user = User.find(params.require(:id))
    if stale?(user)
      response = { success: true,
        result: UserSerializer.new(user, root: false) }
      render json: response , status: :ok
    end
  end
end
```

Listing 5.6 shows an example of how to implement conditional GET caching for a request. Conditional GET caching is a very efficient way of caching as the information is already stored in the client meaning no duplicate information needs to be transferred to the user if it is available in the cache. The performance gains from conditional GET depend on how often information is requested multiple times before it changes. When the cached information is up to date the server only has to verify the freshness of the cached information and respond with a header, no further processing is required for that request.

Custom caching

Ruby on Rails supports low-level custom caching, any information can be stored in the cached and retrieved later. Database models, responses from external services and other results from time consuming processes can be cached and reused for any subsequent requests. When using custom caching the cache has to be kept up to date either by setting an expiration time or by manually checking that the cached information is still up to date. [24]

Caching JSON responses

Generating a JSON response to a request is a resource intensive task that often requires information to be fetched from the database, processed and then converted from a Ruby object to a JSON representation that can be sent to the client. Because caching in RoR allows regular Ruby objects to be stored in the cache it is possible to save the JSON representation of an entire model object in the cache to avoid having to recreate it for any subsequent requests.

Listing 5.7. Code for caching JSON objects

```
class ApplicationSerializer < ActiveModel::Serializer
  delegate :cache_key, :to => :object
```

```
  # Cache entire JSON string
  def to_json(*args)
    Rails.cache.fetch expand_cache_key(self.class.to_s.underscore,
    cache_key, 'to-json') do
      super
    end
  end

  def serializable_hash
    Rails.cache.fetch expand_cache_key(self.class.to_s.underscore,
    cache_key, 'serilizable-hash') do
      super
    end
  end

  ...

  private
  def expand_cache_key(*args)
    ActiveSupport::Cache.expand_cache_key args
  end
end
```

Listing 5.7 shows the code for caching JSON representations of model objects in Ruby on Rails. RoR specific methods `to_json` and `serializable_hash` are overridden to check if information exists in cache before performing the operation. If the information does not exist in the cache then the original methods are called with super, the result from the method call is stored in the cache and returned.

5.3 Ruby versions

The optimized application was benchmarked with CRuby (MRI), JRuby and Rubinius. These implementations of Ruby differ fundamentally in some very important aspects. CRuby uses a GIL to restrict parallel execution of threads and pauses execution of code during the garbage collection phase. JRuby and Rubinius on the other hand do support parallel execution of threads and do not pause execution for garbage collection.

In order to get accurate performance results and to take advantage of the features of the different Ruby versions while still retaining an environment close to real world production environments CRuby was benchmarked using multiple processes. Because the GIL inhibits parallel execution most production environments use multiple server processes each loading their own version of the application into the memory. For optimal performance the application was tested using four processes,

one for each core on the server.

For JRuby and Rubinius on the other hand one single process was used and the number of threads was scaled until no performance gain was observed from increasing the number of threads. JRuby's performance peaked with 48 threads while Rubinius plateaued at 32 threads.

5.4 Garbage collection

Many modern programming languages use a managed memory model. This means that the system is responsible for cleaning up allocated memory after it is no longer needed. In order to find and release memory resources that are not needed by the application a garbage collector is used, the garbage collector scans the memory allocated by the application and frees unused resources. Optimizing the Garbage collector can be a complex task, Ruby makes it possible to supply garbage collection related parameters to tailor the garbage collection after the specific needs of the application. This is often a long process of measuring and fine tuning in order to find anything close to optimal values for the GC settings. Since the settings are dependent on the individual application there are no general rules for how to fine tune garbage collection. TuneMyGC [29] offers a service to make an automated analysis of a Ruby on Rails application running in CRuby. It provides suggestions for custom garbage collection parameters to improve the application performance. Due to time constraints this thesis project did not explore any additional garbage collection optimization methods.

5.5 Other Ruby frameworks

Ruby on Rails is a very complex web framework with functionality to cover most use cases for web applications. This leads to most applications only utilize a small part of all the functionality offered by the framework, [28] discusses different Ruby web frameworks and their performances in a minimal setup. For applications where performance is a huge factor another framework than Ruby on Rails might be more suitable. Switching frameworks requires much less work then rewriting the entire code base in a new programming language. Due to the time constraints of this project and the time consuming process of converting the entire test application to another framework a very simple secondary application was developed in order to be able to benchmark the performance overhead of Ruby on Rails compared to the Ruby micro framework Cuba. [5] The application consists of a very basic API that responds to requests with a JSON string.

Listing 5.8. Minimal API controller in Ruby on Rails

```
class HelloController < ApplicationController
  def hello
    data = { first: 'Hello', last: 'World' }
```

```
    render json: data, status: :ok
  end
end
```

Listing 5.9. Minimal API controller in Cuba

```
require 'cuba'
require 'json'

Cuba.define do
  on get do
    on root do
      data = { first: 'Hello', last: 'World' }
      res.headers["Content-Type"] =
        "application/json; charset=utf-8"
      res.write data.to_json
    end
  end
end
```

Listing 5.8 and 5.9 show the minimal logic for the application API, using these minimal applications the comparative overhead of the frameworks can be evaluated.

Chapter 6

Results

In the previous chapter we introduced different optimization methods for Ruby and the Ruby on Rails frame work. In this chapter we present the results for the optimizations we chose to perform for this thesis project.

6.1 Optimization tools

Multiple tools for measuring and evaluation the performance of Ruby on Rails applications have been presented in earlier chapters of this report. However, not all tools were used for for this project. After comparing the available tools a set of tools for Ruby on Rails performance measurement and optimization was decided on.

6.1.1 Benchmarking and performance measurement

JMeter was used for benchmarking the application with a simulated request load, JMeter is more complex to set up than tools like Apache Bench and Httpress however it supports configuration of advanced simulation plans and offers different metrics for analyzing the results. The system monitoring tool Nmon was used to measure system memory usage during application benchmarks.

6.1.2 Profiling

The application was profiled for CPU and memory usage in CRuby with the ruby-prof gem, it is is one of the most popular Ruby profiling tools and is very simple to use. To be able to use ruby-prof for memory profiling a patched version of CRuby was used. Stack-prof was used as an additional resource for profiling memory usage with CRuyby. As it shows memory usage in number of allocated objects instead of size it makes it possible to find parts of the application that allocates an unnecessary number of objects which may be hard to identify with other profilers.

In order to measure performance of small blocks of code within the application the `assert_performance` gem was developed and used. It measures the execution

time and memory usage of any give section of Ruby code. This was used to measure the impact of small optimizations to isolated parts of the application.

The New Relic RPM performance monitoring gem was used for overall performance measurement in a production like environment. It was used to identify which API endpoints were acting as bottleneck for overall application performance, these endpoints were the main focus for further profiling and optimization.

Profiling was only done with CRuby as the time restrictions of this project did allow for separate profiling and optimization of the application in CRuby, JRuby and Rubinius.

6.2 Basic application optimizations

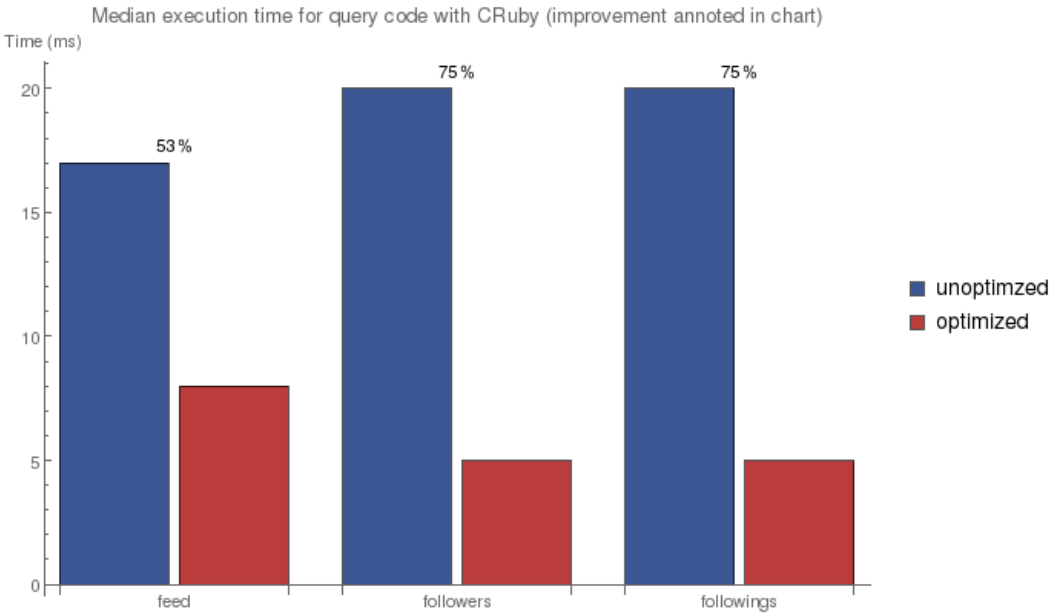

Figure 6.1. SQL query time before and after optimization

By manually selecting information to fetch from the database we were able to minimize the amount of information fetched from the database compared to the default way of querying the database with Active Record. This reduces the application memory allocation and improves application performance. Figure 6.1 shows the difference in SQL query performance before and after optimizing the query logic for different endpoints. By only fetching required information from the database the query execution time was significantly decreased.

The application was then further optimized by adding counter cache columns to the database and by removing unnecessary Ruby on Rails middleware with the `Rails::API` gem. This set of optimizations are denoted as `optimized` in fig-

ures 6.2, 6.3, 6.4 and 6.5. The optimization decreased the response time by 14-36% for the benchmark endpoints of the evaluation application. Figures 6.6 shows the total system memory consumption for the application with different optimization methods. We can see that the optimized application uses almost 300MB less memory than the unoptimized version. By using these simple optimization techniques we were able to not only increase overall performance but also reduce memory consumption significantly.

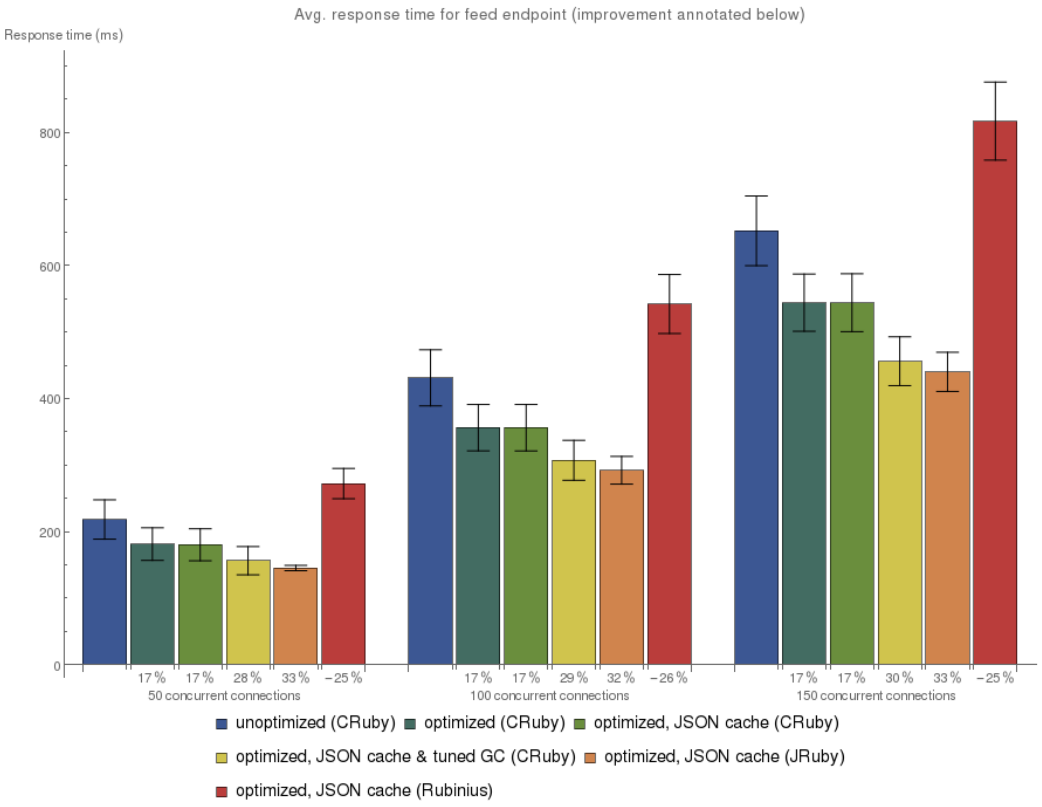

Figure 6.2. Feed response times for different optimizations

6.3 Caching

In addition to the Ruby on Rails built in SQL caching a custom caching layer was implemented to cache the JSON representation of models stored in the database.

The already optimized application was built upon by adding JSON caching. In figures 6.2, 6.3, 6.4 and 6.5 the application with a custom caching layer is denoted as `optimized, JSON cache`. For the users endpoint the JSON caching improves performance significantly with a 96% decrease in response time, the feed endpoint

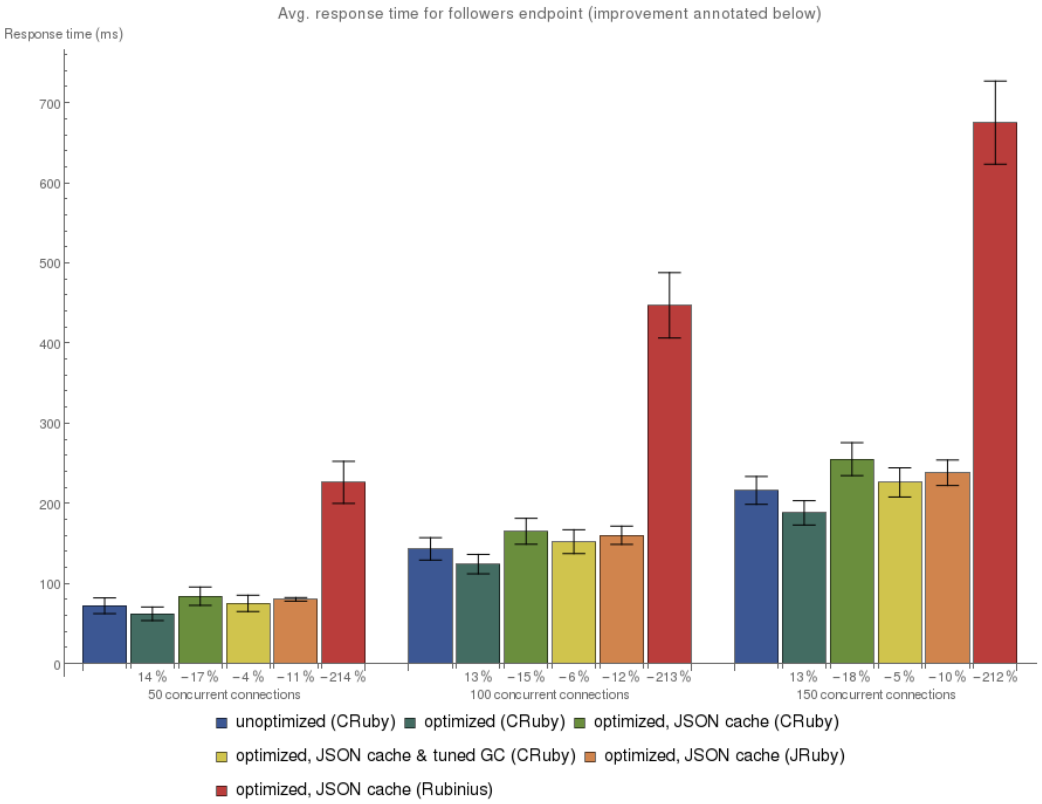

Figure 6.3. Followers response times for different optimizations

however shows unchanged response times with caching whilst the performance of post comments is significantly degraded when JSON caching is introduced.

Caching can improve response time significantly if the operation that produces the results is a time consuming task like in the case of the users endpoint, however the logic required to keep the cache updated comes with a performance penalty and may slow down the application instead of improving performance.

6.4 Ruby versions

The optimized application with JSON caching was evaluated in three different Ruby implementations; CRuby, JRuby and Rubinius. Figure 6.2 shows that JRuby outperforms CRuby noticeably on the feed end point figures however 6.3, 6.4 and 6.5 show only a slight improvement over CRuby on the other endpoints. Much of the performance increase from JRuby comes from the JIT compiler and threading. If the majority of the execution time is spent waiting for the database to perform queries the advantages of JRuby will not make much difference. Rubinius on the

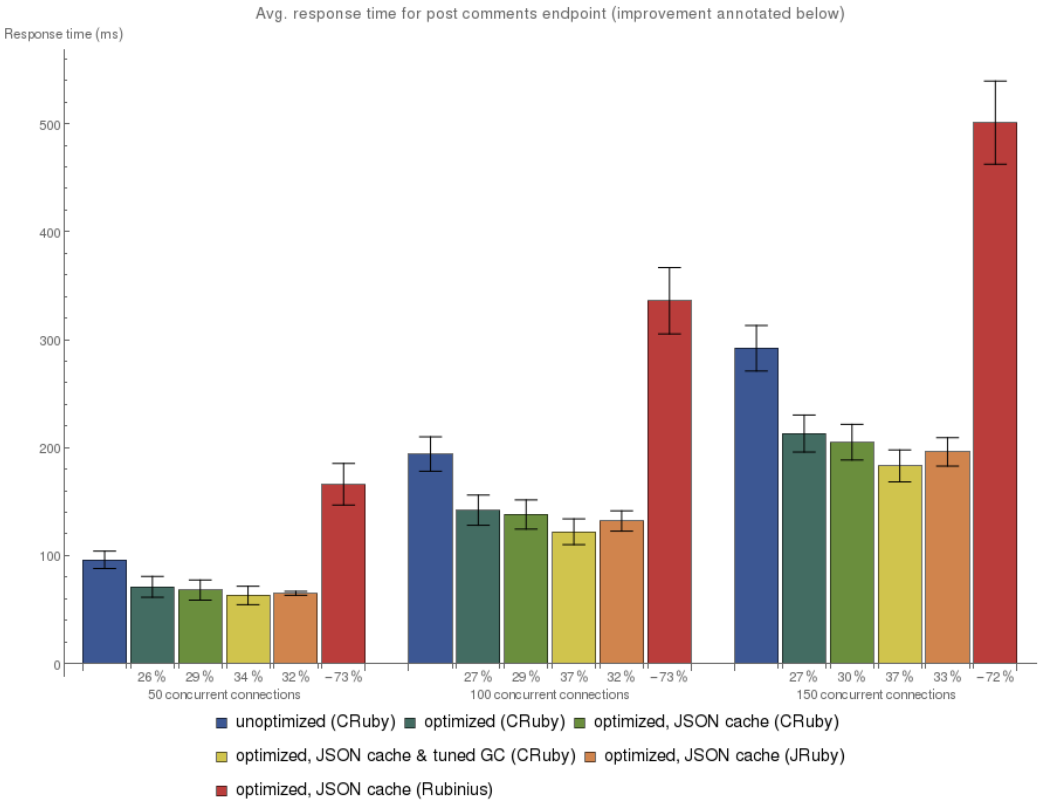

Figure 6.4. Post comments response times for different optimizations

other hand performed significantly worse than the other Ruby implementations in all tests.

Figures 6.6 and 6.7 show the memory consumption for the different Ruby implementations for the feed benchmarking endpoint. JRuby uses significantly more memory than both CRuby and Rubinius. It is easy to see that memory consumption does not increase very steeply with increased server load. A load of 150 concurrent requests just uses a few hundred MB more memory than 50 concurrent requests for all Ruby implementations. Ruby and Ruby on Rails memory consumption does not increase by much as the server load increases unless the number of threads or processes are increased.

6.5 Garbage collection

The optimized application with JSON caching was evaluated with CRuby after tuning the Ruby VM with garbage collection parameters suggested by the automated analysis by TuneMyGC. Figures 6.2, 6.3, 6.4 and 6.2 show the difference in response

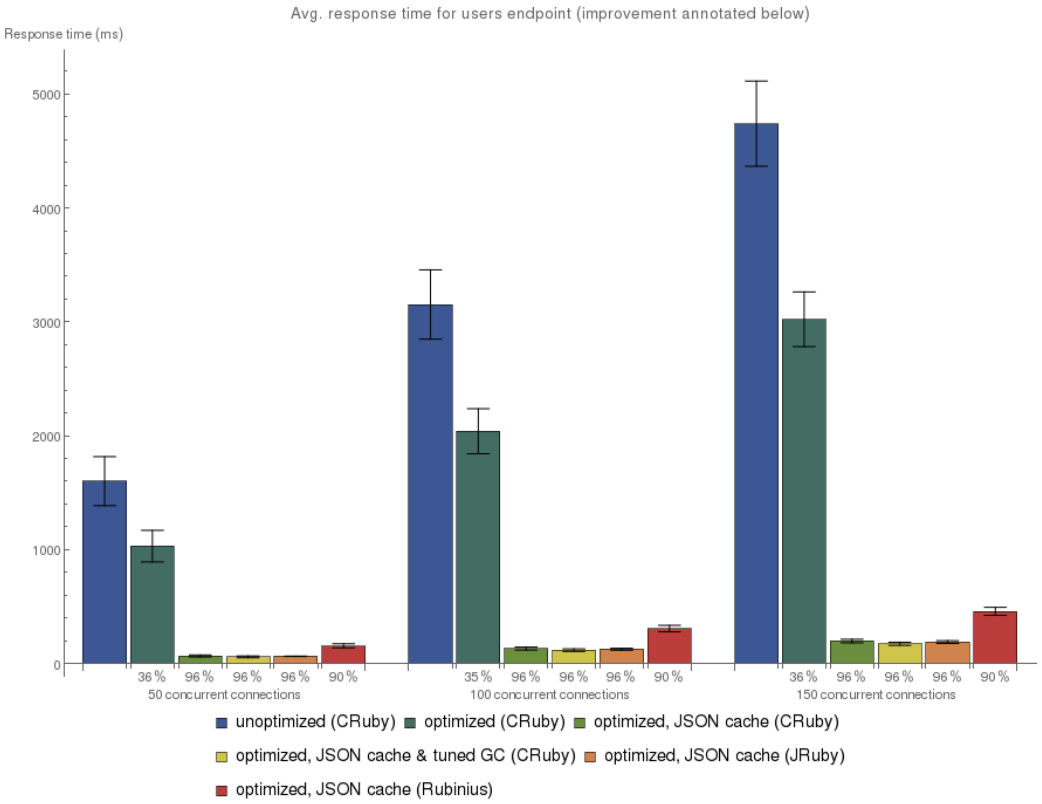

Figure 6.5. Users response times with different optimizations

time after optimizing the garbage collection. Tuning of the garbage collection improved overall application performance for all evaluation endpoints and proved to be a very efficient way of improving performance without having to make any changes to the application code base.

Further manual tuning can improve application performance even more but requires a deeper analysis of the application internals and how they affect garbage collection. TuneMyGC is not available for JRuby and Rubinius but manual tuning can be performed in order to more closely tailor the garbage collection to the application. Thanks to the popularity of the Java JVM and its usage in many performance critical applications optimization of the JVM is a topic that has been extensively researched in the past. Due to time constraints no optimization of the garbage collection was performed on JRuby and Rubinius, but the results from optimization of CRuby garbage collection can be used as an indicator to the efficiency of garbage collection optimizations.

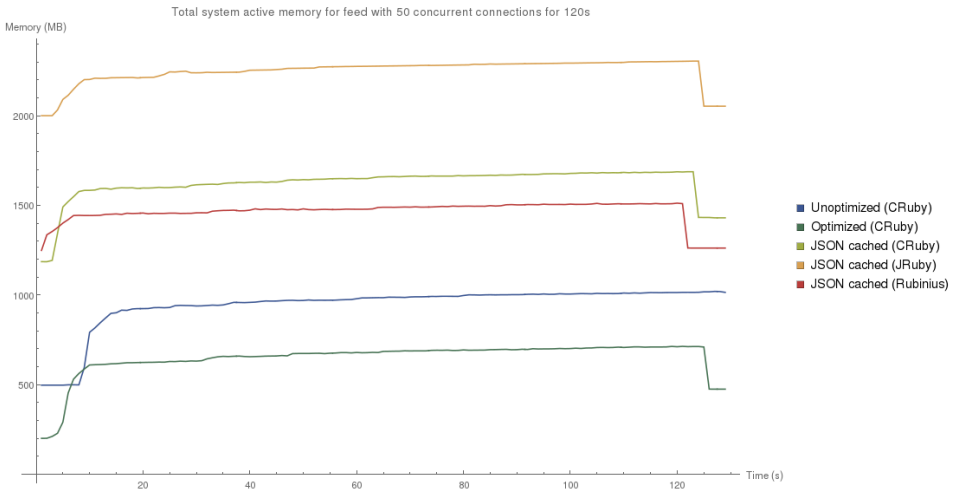

Figure 6.6. Total system memory consumption for the feed endpoint with 50 concurrent connections benchmark. The memory drop after 120 seconds signifies the benchmark ending

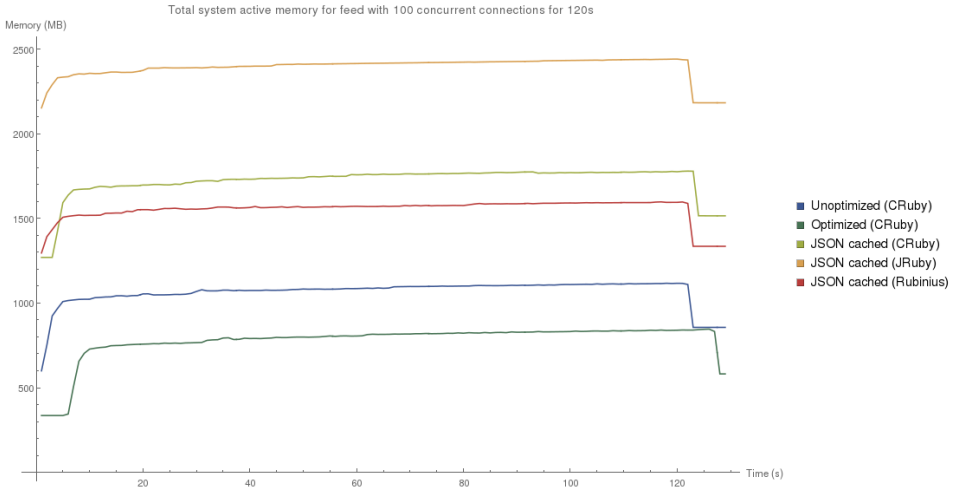

Figure 6.7. Total system memory consumption for the feed endpoint with 150 concurrent connections benchmark. The memory drop after 120 seconds signifies the benchmark ending

6.6 Ruby on Rails server performance

There are many different servers for Ruby, JRuby and Rubinius web applications and the choice of server has large impact on overall application performance. Different Ruby versions perform differently with many of the available servers. All benchmarking was done using the Puma application server [19] because it can be

used with all Ruby versions and can take advantage of multithreading with JRuby and Rubinius as well as multiple processes with CRuby.

When releasing a Ruby on Rails application it is important to take server performance as well as configuration complexity into account. By choosing the correct server for the platform significant performance gains can be achieved. Setting up many different Ruby web application servers correctly is a time consuming process. Due to the time constraints of this thesis project benchmarking was only performed with the Puma server, however [32] benchmarked different Ruby web servers and frameworks and found significant performance difference between the popular web servers with same application. Figure 6.8 shows the throughput of

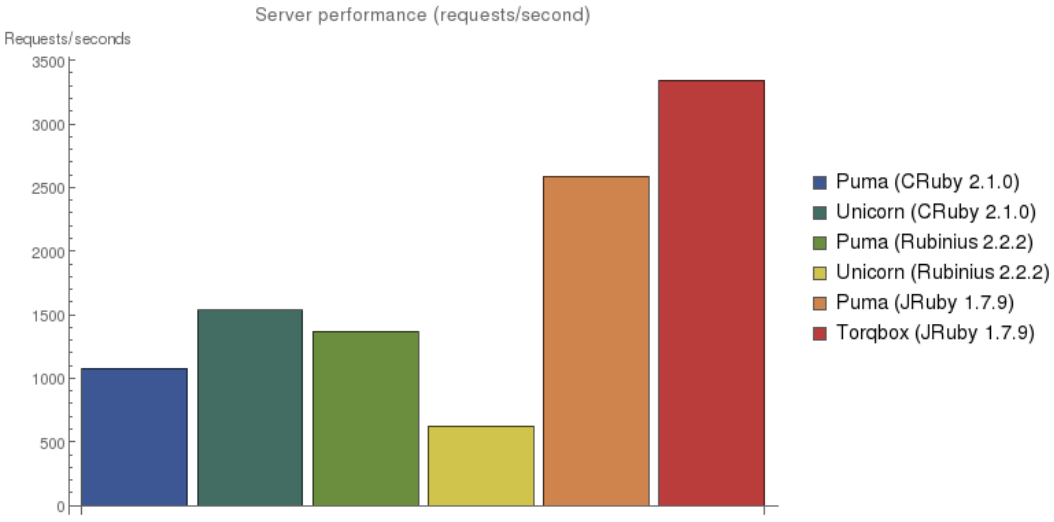

Figure 6.8. Ruby on Rails server performance. Adapted from [32]

different servers for different Ruby versions with a simple Ruby on Rails application. Unicorn performs significantly better than Puma for CRuby, JRuby achieves the best performance with the Torqbox server which significantly outperforms the other servers.

Torqbox is the code name for the development project of Torquebox four [35]. Torquebox is an advanced application server developed specifically for JRuby applications. [36] JRuby with Torqbox provides the highest performance combination for Ruby on Rails applications.

6.7 Other Ruby frameworks

Figure 6.9 shows an 82% lower response time with the Cuba compared to Ruby on Rails with the simple test application. Whilst Ruby on Rails offers much functionality to help develop applications as quickly and painlessly as possible it also

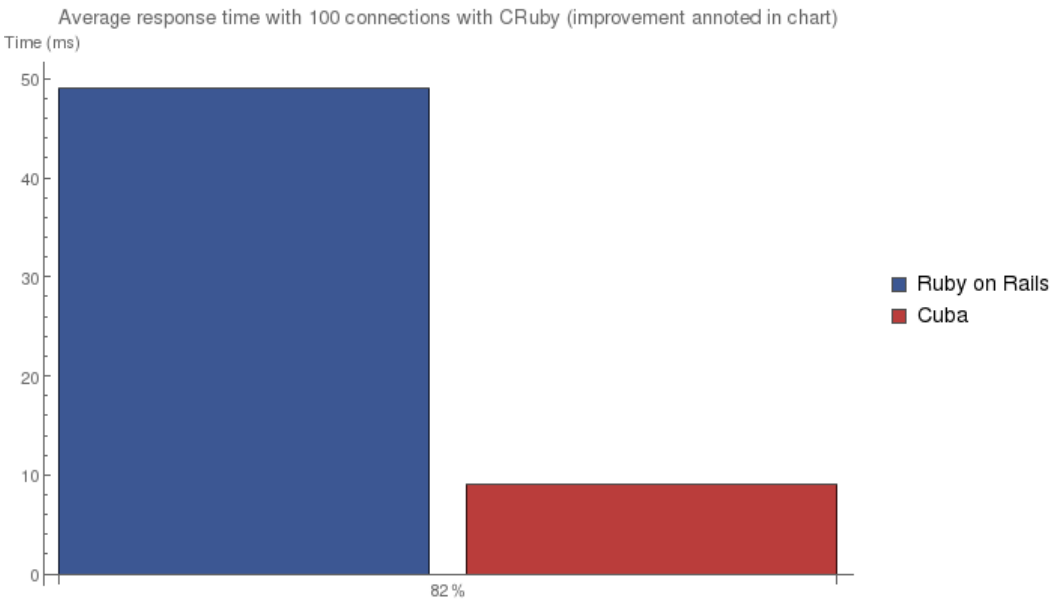

Figure 6.9. Average response time of applications using 100 concurrent request threads over 60 seconds

comes with a significant performance penalty on even the simplest tasks compared to smaller frameworks. Therefore it is important to consider whether the complexity of Ruby on Rails is needed for the application development. If much of the functionality will not be utilized another Ruby framework may be a better fit for the project. Smaller frameworks like Cuba often require more code to be written for the same application functionality but also lowers the overall complexity of the application significantly and thus makes it possible to create better performing applications.

Figure 6.10 shows the total system memory consumption for running Ruby on Rails and Cuba both in idle and with 100 concurrent requests over 60 seconds. Ruby on rails has significantly higher memory consumption in both idle and under stress. For both frameworks the application server Puma was used to create four server processes with four threads each using CRuby version 2.2.2. Cuba uses significantly less memory under heavy load and seems to plateau at just under 500 MB total system memory consumption while Ruby on Rails produces close to 700 MB memory consumption. The Cuba framework performs significantly better while using less memory under heavy load. These benchmarks were performed with a very simple application and the results may differ for larger applications however they show that Ruby web applications have the potential to perform well.

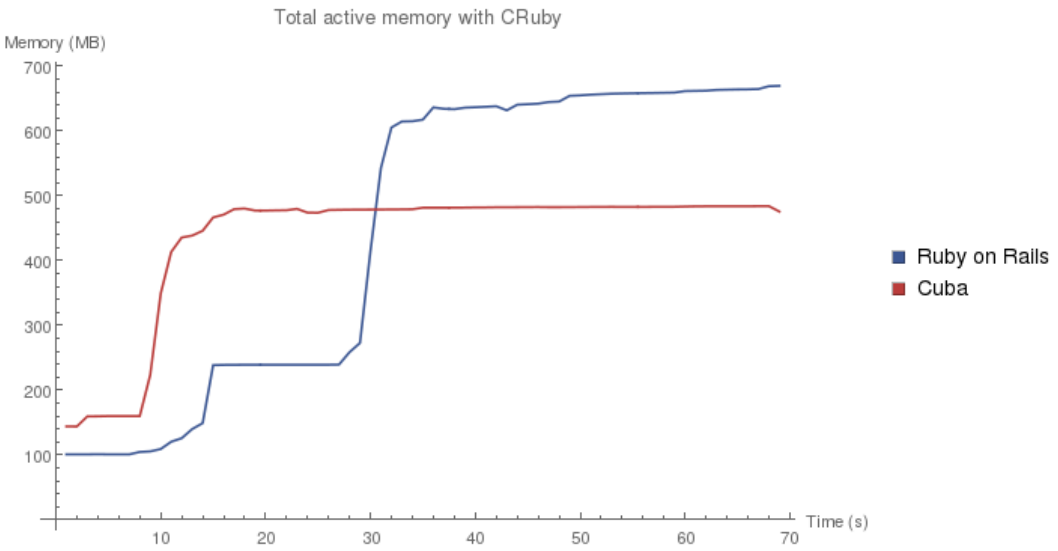

Figure 6.10. Total system memory consumption for Ruby on Rails and Cuba going from idle to 100 concurrent requests

Chapter 7

Discussion and conclusion

This chapter is dedicated to discussing the results presented in the previous chapter and the conclusions that can be drawn from this project. This thesis has described how the Ruby on Rails web framework can be optimized to improve overall application performance and make it easier to scale an application.

This thesis focuses on how to improve performance of an existing Ruby on Rails application rather than how to start a new project. When developing a new product it is often vital to be as quick as possible to develop and release a market ready product, this is where Ruby on Rails shines. Performance and scaling is usually something to deal with later in the product life cycle if and when the product becomes popular enough. Due to the fact that many new products fail very early it is often better to deal with performance problems at a later if they arise.

In order to increase performance we looked at different ways of optimizing Ruby from ways of writing more efficient code, slimming down Ruby on Rails, caching, garbage collection to using alternative Ruby versions and the performance of different servers.

7.1 Writing efficient code

A developer should always strive to write the most efficient code possible. Ruby on Rails however, abstracts away much of the database interaction with `Active Record` making it so that developers do not even have to write any database queries but can instead interact with the database using pure Ruby. This makes querying the database very easy but the database specific queries that are generated are often suboptimal performance wise. By writing queries that only fetch the minimum required information from the database we saw significant increases in query performance. With this in mind we propose a change in the mindset of how to interact with the database in the Ruby on Rails community. It is very easy to just view the database as an object store where your models are persisted and retrieved. However, going back to the old mindset of tables and columns makes it easier to realize exactly what information is needed for a certain task and query for only that

information instead of fetching and instantiating all the relevant model objects.

Ruby is not very fast compared to many other high performance programming languages. It was never meant for writing time sensitive calculation heavy operations and should not be used for such a thing. Luckily it is easy to incorporate code from other languages in a Ruby on Rails application, CRuby has great support for writing high performing extensions in C that can be called from the Ruby code. JRuby has the added advantage of being able to use Java code wherever needed throughout the application thus combining the fast development speed of the dynamic programming language Ruby with the high performance of the static language Java.

7.2 Increasing the performance of the application

When increasing the performance of an existing application it is important to take into account how much change it requires to the existing code base. We saw that choosing the best application server can improve performance significantly without requiring any change to the application code, simple optimization of the garbage collector also gave a noticeable improvement in application performance.

A simple yet efficient way of increasing performance is to introduce caching, caching can improve performance immensely if the results of time consuming operations can be cached and reused. However, caching adds the additional complexity of making sure that the cached information is always up to date and for some of the benchmark endpoints this proved to deteriorate application performance instead of improving it. It was unexpected that caching can have a significantly negative effect on performance. Sometimes it is quicker to fetch information from the database and create the response than making sure that the cached information is up to date and fetching it from cache. It is important to analyze for which operations it makes sense to cache and for which it does not. Caching can be done in layers from entire responses down to single database queries giving multiple opportunities to find information in the cache without having to perform the entire request.

JRuby offers the best performance of the evaluated Ruby implementations but it also requires a significant change to the application in order to make it compatible. Any Ruby gem that use C extensions for better performance is incompatible with JRuby. For many popular gems there are Java implementations that can be used instead. If however, the application relies on any incompatible gems that lack Java based alternatives the functionality may have to be re-implemented in order to make the application work with JRuby and this can be a very time consuming process. The performance gains from switching to JRuby may out weight the conversion complexity if performance is a big issue.

That JRuby would outperform CRuby was expected. The JVM is famous for its high performance and the Ruby MRI is known to have performance issues. The performance results of Rubinius on the other hand were not expected. On paper Rubinius looks like a high performance Ruby implementation with its JIT compila-

tion, advanced concurrent garbage collection and support for parallel execution in system level threads. The results from the benchmarks surprisingly show that Rubinius performed much worse than both CRuby and JRuby. Time constrains for this project did not allow for further investigation into what caused these performance problems with Rubinius.

7.3 Better scaling of the application

When we talk about scaling in this thesis we talk about scaling the application to cater to more users on a single server, this differs from the common definition of scaling as adding more servers. Distributing the application to multiple servers is something that is not handled by the Ruby on Rails framework and is considered outside the scope of this thesis.

One big problem with scaling a CRuby application is that the GIL lock prevents parallel execution of Ruby code, in order to achieve true parallelism multiple processes have to be used, this means that the entire Ruby on Rails framework and gems have to be loaded into memory multiple times since each process needs its own copy. A big problem with process based scaling is that it requires much more memory. JRuby and Rubinius on the other hand supports real threading which allows each thread to share one singe instance of the Ruby on Rails framework in memory. The application can now be scaled by increasing the number of threads and because the memory increase with each thread is minimal it is easier to get the best performance possible from the available server hardware.

7.4 Other Ruby frameworks

Ruby on Rails includes rather large overhead. It is possible to reduce it by removing unused middleware but Ruby on Rails will never be a lean and fast framework. If performance is a key priority then switching to a smaller and better performing framework can be a good idea. The Cuba framework showed almost five times faster average response time for a simple hello world application. This shows how much overhead Ruby on Rails suffers from when it can be considered slow even for such simple applications. Starting out with a small high performing framework and only adding features that are needed for the application is a good way of designing a high performance Ruby application. While this may be a costly process for an existing application it is less work than rewriting the entire application in a different higher performing programming language.

Cuba was expected to perform significantly better than Ruby on Rails because it is a very small framework that comes with much less functionality and overhead than Ruby on Rails. However, the fact that even for very simple applications that only use the most basic functionality of Ruby on Rails the Cuba framework still managed to produce an 82% decrease in response time as well significantly lower total system memory usage indicates serious performance problems with the Ruby

on Rails frameworks. Maybe smaller more modular frameworks are preferable to larger complex systems like Ruby on Rails when it comes to making the compromise between performance and provided functionality.

7.5 Conclusion

After performing optimizations, benchmarks and evaluations of Ruby on Rails we should conclude whether we were able to achieve the goals set up by our scientific questions to determine if and how Ruby on Rails can be optimized for performance and scalability.

7.6 What tools can be used to evaluate Ruby on Rails performance and scalability?

In this project we evaluated many tools to measure and evaluate the performance and found that the JMeter benchmarking application works well for benchmarking applications to get a sense of how well it performs under severe stress. Because performance usually becomes a problem when the application is under heavy stress this is a very valuable tool for evaluation of performance in a worst case scenario. JMeter can also be used to measure the scalability of the application, by gradually increasing the load it possible to get an idea how the application scales with increased load.

The New Relic RPM performance monitoring gem provides very valuable information as to which parts of the application are performance bottlenecks. Its light weight overhead makes it possible to use it in a production system in order to gain real world performance information. When a bottleneck has been localized then Ruby compatible profilers like `ruby-prof` can be used to optimize the functionality that is slowing the application down.

In order to reliably measure the execution time and memory consumption for an arbitrary block of ruby code the `assert_performance` gem was developed. It makes it easy to measure and log the performance of any piece of Ruby code to track the efficiency performed optimizations.

7.6.1 Does Ruby implementation impact performance?

From our evaluation of the performance of CRuby, JRuby and Rubinius we found that the Ruby implementation has a large impact on application performance. JRuby produced the best performance of all the Ruby implementations and was significantly faster than the others in some cases. The fact that JRuby makes it easy to write performance dependent parts of the application in Java and call them in the Ruby application makes an even bigger case for using JRuby over CRuby for applications that require higher performance.

7.6.2 What parts of Ruby on Rails can be optimized?

Ruby on Rails is a very large framework with many different aspects that may optimized for better performance. In order to stay within the project time frame we made a choice to focus on optimizing certain parts of the framework.

Memory consumption

Ruby on Rails memory consumption can be optimized to a large degree by writing efficient application code. All models fetched from the database have to be instantiated as Ruby objects and this can use substantial amounts of memory. By only fetching the information that is needed memory usage can be decreased. Our optimized application used almost 300 MB less total system memory than the unoptimized version.

Object-relational mapping (ORM)

The built in ORM in Ruby on Rails is Active Record, it provides a convenient way of storing and retrieving model objects from the database without having to write database specific queries. This is very convenient as the underlying database system can be changed without having to rewrite any of the code base. The database can be queried using Ruby code which is converted to database specific queries by Active Record. Unless manually specified the generated queries tend to fetch too much information from the database. We were able to decrease query execution time by 75% by optimizing the queries to only the information required for each operation. The Active Record ORM can be optimized for better performance, by analyzing the queries generated by Active Record it is easy to find opportunities for optimization.

Garbage collection

We used `TuneMyGC` to perform an automated analysis of the application in order to get suggestions for better tuned garbage collection parameters. Using the suggested parameters increased performance in all benchmarks. This improvement was gained without any manual tuning, only using the suggested parameters from `TuneMyGC`, further manual tuning could possibly increase the performance even more. We did not perform any garbage collector tuning for JRuby and Rubinius due to time constraints but from our results we can conclude that optimizing the garbage collection can improve performance substantially without requiring any change to the application code base.

Choice of application server

Due to the complexity and time consumption of configuring multiple Ruby web application servers for a production environment we did not benchmark the performance of different Ruby servers, however related work shows that the application server has a very large impact on overall application performance. JRuby with a

correctly configured Torqbox server performs substantially better than other Ruby servers. We can conclude that performance can be improved by using the best performing application server available for the chosen production environment.

Removal of unnecessary modules

The `Rails::API` gem was benchmarked as part of the optimized application suite and not benchmarked on its own, however related work shows that using the slimmed down `Rails::API` middleware stack decreases application response time and memory consumption. By removing unnecessary middleware we can retain all the required functionality yet improve performance. For any API application we recommend either using the `Rails::API` gem or manually removing all middleware related to the view in Ruby on Rails.

7.6.3 Verdict

Ruby on Rails can be made significantly faster with optimization and using JRuby will both increase performance and make scaling the application much easier. However, Ruby on Rails is not a framework that should be used for performance critical applications, Ruby on Rails is big and carries large overhead and will therefore never be a fast framework compared to some of its leaner counterparts.

Ruby on Rails is great at what it was developed for, a general purpose web application framework with focus on developer productivity. It can be optimized to be fast enough for most production requirements but if performance is critical to the application then we propose the use of a different framework. The Cuba micro framework was almost five times faster than Ruby on Rails for a simple hello world application and is a good starting point for development of a high performing Ruby web application.

7.6.4 A high performance Ruby on Rails application

In order to achieve better performance with a Ruby on Rails web application we propose

- Remove any unnecessary middleware from Rails, any modules that are not needed by the application is a waste of resources.
- Do not fetch unnecessary information from the database. Models fetched from the database need to be instantiated and later on garbage collected and this consumes system resources.
- Use JRuby instead of regular CRuby for real threading and to take advantage of the high performance of the Java JVM.
- Write calculation heavy functionality in Java which can be called from the JRuby application in order to take full advantage of the high performance of the Java programming language.

- Use the Torqbox JRuby server, benchmarks show that Torqbox is by far the highest performing JRuby web application server.
- Optimize the garbage collector either by automated analysis or manual tuning.
- Cache results from expensive operations if they can be reused for subsequent requests. Fetching information from the cache can be multiple orders of magnitude faster than performing an expensive operation so make sure to reuse results where possible.

7.7 Future work

In this section we present some ideas for future work based on the results of this thesis. For future work we suggest a broader analysis of Ruby web frameworks. Ruby on Rails may be the most popular framework but there are many other competent frameworks available. One suggested topic is to analyze and compare the performance and developer productivity between different popular frameworks. There may be frameworks which offer close to the same developer productivity as Ruby on Rails but with better performance. Such a framework would be ideal for developing applications that need to hit the market as quickly as possible but still needs reasonable performance and scaling possibilities.

Another interesting idea is to evaluate how static typing would affect overall Ruby performance. There are discussions on whether or not to implement an optional static typing system for future versions of Ruby and this presents a good opportunity for a deeper analysis of how static typing would affect Ruby performance. A static type system makes it easier for the interpreter to make optimizations as it knows which type the variable is going to be and can optimize performance using this information whereas in a dynamic typing system the interpreter has to account for that the variable type can change.

As of writing this report a new major version of JRuby was released, the JRuby 9.0.0.0 features many new concepts and performance improvements. A deeper analysis of this new version is suggested as future work. With this new version maybe Ruby on Rails and other Ruby web frameworks can perform even better.

Bibliography

[1] Active record github. https://github.com/rails/rails/tree/master/activerecord. [Accessed 28 June 2015].

[2] Amazon aws website. http://aws.amazon.com/. [Accessed 28 June 2015].

[3] Apache bench documentation. https://httpd.apache.org/docs/2.2/programs/ab.html. [Accessed 28 June 2015].

[4] Chef website. https://www.chef.io. [Accessed 28 June 2015].

[5] Cuba ruby framework website. http://cuba.is. [Accessed 01 July 2015].

[6] Httpress bitbucket repository. https://bitbucket.org/yarosla/httpress. [Accessed 28 June 2015].

[7] Instagram website. https://instagram.com. [Accessed 28 June 2015].

[8] Jmeter website. http://jmeter.apache.org/. [Accessed 28 June 2015].

[9] Jprofiler product website. https://www.ej-technologies.com/products/jprofiler/overview.html. [Accessed 28 June 2015].

[10] Jruby website. http://jruby.org. [Accessed 28 June 2015].

[11] Json website. https://www.json.org. [Accessed 28 June 2015].

[12] Kcachegind. http://kcachegrind.sourceforge.net/html/Home.html. [Accessed 28 June 2015].

[13] A theory temporal and spatial locality. http://snir.cs.illinois.edu/PDF/TemporalandSpatialLocality.pdf. [Accessed 28 June 2015].

[14] Mongodb website. https://www.mongodb.org. [Accessed 28 June 2015].

[15] New relic rpm github. https://github.com/newrelic/rpm. [Accessed 28 June 2015].

[16] Nmon website. http://nmon.sourceforge.net/pmwiki.php, . [Accessed 28 June 2015].

[17] Nmonvisualizer website. http://nmonvisualizer.github.io/nmonvisualizer/, . [Accessed 28 June 2015].

[18] Parse. https://www.parse.com/. [Accessed 28 June 2015].

[19] Puma application server. http://puma.io/. [Accessed 18 August 2015].

[20] Ruby on rails website. http://rubyonrails.org/, . [Accessed 28 June 2015].

[21] Rails api gem github. https://github.com/rails-api/rails-api, . [Accessed 12 August 2015].

[22] Json api with rails-api and active model serializers. http://adamniedzielski.github.io/blog/2014/03/02/json-api-with-rails-api-and-active-model-serializers/, . [Accessed 12 August 2015].

[23] Rails api to be part of rails 5. http://wyeworks.com/blog/2015/4/20/rails-api-is-going-to-be-included-in-rails-5/, . [Accessed 12 August 2015].

[24] Rails caching documenation. http://edgeguides.rubyonrails.org/caching_with_rails.html, . [Accessed 19 August 2015].

[25] Rubinius website. http://rubini.us, . [Accessed 28 June 2015].

[26] Ruby on rails with rubinius. http://rubini.us/doc/en/guides/migrating-from-mri-to-rubinius, . [Accessed 28 June 2015].

[27] Ruby-prof. https://github.com/ruby-prof/ruby-prof, . [Accessed 28 June 2015].

[28] Made by market blog. http://www.madebymarket.com/blog/dev/ruby-web-benchmark-report.html, . [Accessed 01 July 2015].

[29] Tunemygc service. https://tunemygc.com, . [Accessed 12 August 2015].

[30] Rubyjmeter github. https://github.com/flood-io/ruby-jmeter, . [Accessed 28 June 2015].

[31] Ruby website. http://www.ruby-lang.org, . [Accessed 28 June 2015].

[32] Benchmark of different ruby web servers and frameworks. http://www.madebymarket.com/blog/dev/ruby-web-benchmark-report.html, . [Accessed 18 August 2015].

[33] Slagkryssaren website. http://slagkryssaren.com. [Accessed 28 June 2015].

[34] Stackprof github. https://github.com/tmm1/stackprof. [Accessed 28 June 2015].

[35] Torquebox blog. http://torquebox.org/news/2014/07/01/torquebox-4-update/, . [Accessed 18 August 2015].

[36] Torquebox website. http://torquebox.org/, . [Accessed 18 August 2015].

[37] Yourkit product website. https://www.yourkit.com. [Accessed 28 June 2015].

[38] Javier Alcázar Zapién. Debugging parallel programs using fork handlers. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 112–121, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3404-4. doi: 10.1145/2712386.2712390. URL http://doi.acm.org.focus.lib.kth.se/10.1145/2712386.2712390.

[39] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375586. URL http://doi.acm.org/10.1145/1375581.1375586.

[40] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685.

[41] Paul Deitel and Harvey Deitel. *C++ How to Program*. Prentice Hall Press, Upper Saddle River, NJ, USA, 8th edition, 2011. ISBN 0132662361, 9780132662369.

[42] Alexander Dymo. *Ruby Performance Optimization: Why Ruby Is Slow and How to Fix It*. Pragmatic Bookshelf, 2015.

[43] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542528. URL http://doi.acm.org/10.1145/1543135.1542528.

[44] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254.

[45] David Geer. Will software developers ride ruby on rails to success? *Computer*, 39(2):18–20, February 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.74. URL http://dx.doi.org/10.1109/MC.2006.74.

[46] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.

[47] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4): 18–21, December 1990. ISSN 0163-5964. doi: 10.1145/121973.121975. URL http://doi.acm.org/10.1145/121973.121975.

[48] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43 (2):12–14, February 2010. ISSN 0018-9162. doi: 10.1109/MC.2010.58. URL http://dx.doi.org/10.1109/MC.2010.58.

[49] Yishan Li and S. Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19, Aug 2013. doi: 10.1109/PACRIM.2013.6625441.

[50] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

[51] Remigius Meier and Armin Rigo. A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, pages 4:1–4:4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2914-9. doi: 10.1145/2633301.2633305. URL http://doi.acm.org/10.1145/2633301.2633305.

[52] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 1st edition, 2011. ISBN 9781934356654.

[53] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. Eliminating global interpreter locks in ruby through hardware transactional memory. *SIGPLAN Not.*, 49(8):131–142, February 2014. ISSN 0362-1340. doi: 10.1145/2692916.2555247. URL http://doi.acm.org/10.1145/2692916.2555247.

[54] Zachary Parker, Scott Poe, and Susan V. Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1901-0. doi: 10.1145/2498328.2500047. URL http://doi.acm.org/10.1145/2498328.2500047.

[55] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails 4*. Pragmatic Bookshelf, 4th edition, 2013. ISBN 1937785564, 9781937785567.

[56] Koichi Sasada. Yarv: Yet another rubyvm: Innovating the ruby interpreter. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 158–159, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. doi: 10.1145/1094855.1094912. URL http://doi.acm.org/10.1145/1094855.1094912.

[57] Masatoshi Seki. druby and rinda: Implementation and application of distributed ruby and its parallel coordination mechanism. *Int. J. Parallel Program.*, 37(1):37–57, February 2009. ISSN 0885-7458. doi: 10.1007/s10766-1108-0086-1. URL http://dx.doi.org/10.1007/s10766-1108-0086-1.

[58] Pat Shaughnessy. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. No Starch Press, San Francisco, CA, USA, 2013. ISBN 1593275277, 9781593275273.

[59] Alan Jay Smith. *Cache memories*, volume 14. 1982.

[60] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53 (4):10–11, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721659. URL http://doi.acm.org/10.1145/1721654.1721659.

[61] Jesse Storimer. *Working with Ruby Threads*. Pragmatic Bookshelf, 2013.

[62] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001. ISBN 0130313580.

[63] Dave Thomas, Andy Hunt, and Chad Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2013. ISBN 1937785491, 9781937785499.

[64] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 278–287, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772993. URL http://doi.acm.org/10.1145/1772954.1772993.

[65] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09, pages 79–88, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: 10.1145/1640134.1640147. URL http://doi.acm.org/10.1145/1640134.1640147.

Appendix A

Code

A.1 Assert performance gem

```
require "assert_performance/version"
require "benchmark"
require "parse-ruby-client"
require 'rubygems'
require "active_record"

##
# Benchmarks code and database calls and optionally submits
    results to a Parse database
#
# Modified from example in "Ruby Performance Optimization:
    Why Ruby Is Slow and How To Fix It"
# written by Alexander Dymo
#
#
module AssertPerformance

  class PerformanceTestTransactionError < StandardError
  end

  def self.benchmark_code(name, &block)
    operation_results = nil
    read, write = IO.pipe
    (0..30).each do |i|
      # Force GC to reclaim all memory used in previous run
      GC.start

      pid = fork do
```

```
    # GC extra memory that fork allocated
    GC.start
    # Disable GC if option set
    GC.disable if ENV["RUBY_DISABLE_GC"]

    # Store results in a between runs
    benchmark_results = File.open("benchmark_results_#{
        name}", "a")
    elapsed_time, memory_after, memory_before = nil
    begin
      ActiveRecord::Base.transaction do
        memory_before = `ps -o rss= -p #{Process.pid}`.
            to_i
        elapsed_time = Benchmark::realtime do
          operation_results = yield
        end
        memory_after = `ps -o rss= -p #{Process.pid}`.
            to_i
        raise PerformanceTestTransactionError
      end
    rescue PerformanceTestTransactionError
      # Rollback database
    end
    # Skip first run to exclude cold start measurements
    if i > 0
      # Store runtime
      benchmark_results.puts elapsed_time.round(6)
    end
    benchmark_results.close
    GC.enable if ENV["RUBY_DISABLE_GC"]

    read.close
    results = {results: operation_results , memory: (
        memory_after - memory_before)}
    Marshal.dump(results,write)
  end
  Process::waitpid pid
end
measurements = File.readlines("benchmark_results_#{name}
    ").map do |value|
  value.to_f
end
File.delete("benchmark_results_#{name}")
```

```
    average = average(measurements).round(5)
    stddev = standard_deviation(measurements).round(5)

    # If parse object is set store results in parse database
        for further analysis
    id = nil
    if @parse
      parse_benchmark = Parse::Object.new("CodeBenchmark")
      parse_benchmark['time'] = Time.new
      parse_benchmark['name'] = name
      parse_benchmark['average'] = average
      parse_benchmark['standard_deviation'] = stddev
      parse_msg = parse_benchmark.save
      puts "Saving data to Parse: #{parse_msg}"
      id = parse_benchmark["objectId"]
    end
    # Return benchmark and operation results so they can be
        validated
    write.close
    process_results = read.read
    processed_results = Marshal.load(process_results)
    return {
      results: processed_results[:results],
      benchmark: {
        name: name,
        average: average,
        standard_deviation: stddev,
        memory: processed_results[:memory],
        id: id
      }
    }
  end

  def self.benchmark_database(name)
    result = []
    ActiveSupport::Notifications.subscribe "sql.
        active_record" do |*args|
      event = ActiveSupport::Notifications::Event.new(*args)
      query_name = event.payload[:sql]
      next if ['SCHEMA'].include?(query_name)
      result << query_name
    end
    yield
    ActiveSupport::Notifications.unsubscribe("sql.
```

```
        active_record")
      id = nil
      if @parse
        parse_benchmark = Parse::Object.new("DatabaseBenchmark
            ")
        parse_benchmark['name'] = name
        parse_benchmark['queries'] = result
        parse_msg = parse_benchmark.save
        puts "Saving data to Parse: #{parse_msg}"
        id = parse_benchmark["objectId"]
      end
      # Put results into hash under benchmark to match
          benchmark_code structure
      { benchmark: {name: name, queries: result, id: id} }
    end

    def self.standard_deviation(measurements)
      Math.sqrt(measurements.inject(0){|sum, x| sum + (x -
          average(measurements)) ** 2}.to_f / (measurements.
          size - 1))
    end

    def self.average(measurements)
      measurements.inject(0) { |sum, x| sum + x }.to_f /
          measurements.size
    end

    def self.setup_parse(parse_details)
      Parse.init(parse_details)
      @parse = true
      puts "Setting up parse"
    end
  end
```

A.2 Application

A.2.1 Models

User

```
class User < ActiveRecord::Base
  has_secure_password
  validates :username, :email, presence: true, uniqueness:
      true
```

```
  validates :birthdate, :description, :gender, presence:
      true
  validates :password, length: { minimum: 6 }, on: :create

  has_many :user_tags
  has_many :posts, foreign_key: 'author_id'
  has_many :likes
  has_many :tagged_posts, through: :user_tags, class_name: '
      Post', source: :post

  has_many :user_followings
  has_many :followings, through: :user_followings

  has_many :inverse_user_followings, class_name: '
      UserFollowing', foreign_key: 'following_id'
  has_many :followers, through: :inverse_user_followings,
      source: :user

  # Check username and password, if they match
  # generate a globally unique token and return it
  def self.authenticate(username, password)
    user = self.find_by(username: username)
    if user && user.authenticate(password)
      begin
        token = SecureRandom.hex
      end while self.find_by(token: token)
      user.token = token
      user.save!
      token
    end
  end

  # If correct accesstoken is given return user else nil
  def self.authenticate_with_token(token)
    User.find_by(token: token)
  end

  def active_model_serializer
    UserSerializer
  end

  # Follow a user only if we aren't following him/her
      already
  def follow(user_to_follow)
```

```
    if(self.id != user_to_follow.id && self.followings.where
       (id: user_to_follow.id).count == 0)
      self.followings << user_to_follow
      self.save!
    else
      false
    end
  end

end
```

User following

```
class UserFollowing < ActiveRecord::Base
  belongs_to :user, touch: true, counter_cache: :
     followings_count
  belongs_to :following, class_name: 'User', touch: true,
     counter_cache: :followers_count
end
```

User tag

```
class UserTag < ActiveRecord::Base
  belongs_to :post, touch: true
  belongs_to :user, touch: true
  belongs_to :comment, touch: true
end
```

Tag

```
class Tag < ActiveRecord::Base
  validates :text, presence: true, uniqueness: true
  has_and_belongs_to_many :posts, touch: true
  has_and_belongs_to_many :comments, touch: true
end
```

Post

```
class Post < ActiveRecord::Base
  include Taggable
  validates :image, :description, presence: true

  belongs_to :author, class_name: 'User', touch: true
  has_and_belongs_to_many :tags, touch: true
  has_many :user_tags
  has_many :comments
```

```
  has_many :likes
  has_many :tagged_users, through: :user_tags, class_name: '
      User', source: :user

  mount_base64_uploader :image, ImageUploader
end
```

Like

```
class Like < ActiveRecord::Base
  belongs_to :user, touch: true
  belongs_to :post, touch: true

  def self.like(user, post)
    if(Like.where(user: user, post: post).count == 0)
      Like.create!(user: user, post: post)
    else
      false
    end
  end
end
```

Comment

```
class Comment < ActiveRecord::Base
  include Taggable
  validates :comment, presence: true
  belongs_to :author, class_name: 'User', touch: true
  belongs_to :post, touch: true
  has_and_belongs_to_many :tags, touch: true
  has_many :user_tags
  has_many :tagged_users, through: :user_tags, class_name: '
      User', source: :user
end
```

Concerns

```
module Taggable
  extend ActiveSupport::Concern
  # Sets up correct associations for a post
  # @param {Array} tags - Tags for the post
  # @param {Array} users_tagged - Users to be tagged in the
      post
  def create_assoc_and_save(tags = [], users_tagged = [])
    tags = [tags] unless tags.respond_to?('each')
```

```
    users_tagged = [users_tagged] unless users_tagged.
       respond_to?('each')
    self.tags = tags
    user_tag_args = {}
    users_tagged.each do |user|
      user_tag_args[self.class.name.downcase.to_sym] = self
      user_tag_args[:user] = user
      UserTag.create!(user_tag_args)
    end
    self.save!
    self.id
  end
end
```

Uploaders

```
# encoding: utf-8

class ImageUploader < CarrierWave::Uploader::Base

  # Include RMagick or MiniMagick support:
  # include CarrierWave::RMagick
  # include CarrierWave::MiniMagick

  # Choose what kind of storage to use for this uploader:
  storage :file
  # storage :fog

  # Override the directory where uploaded files will be
     stored.
  # This is a sensible default for uploaders that are meant
     to be mounted:
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{
       model.id}"
  end

  # Provide a default URL as a default if there hasn't been
     a file uploaded:
  # def default_url
  #   # For Rails 3.1+ asset pipeline compatibility:
  #   # ActionController::Base.helpers.asset_path("fallback
     /" + [version_name, "default.png"].compact.join('_'))
  #
```

```
  #   "/images/fallback/" + [version_name, "default.png"].
      compact.join('_')
  # end

  # Process files as they are uploaded:
  # process :scale => [200, 300]
  #
  # def scale(width, height)
  #   # do something
  # end

  # Create different versions of your uploaded files:
  # version :thumb do
  #   process :resize_to_fit => [50, 50]
  # end

  # Add a white list of extensions which are allowed to be
      uploaded.
  # For images you might use something like this:
  # def extension_white_list
  #   %w(jpg jpeg gif png)
  # end

  # Override the filename of the uploaded files:
  # Avoid using model.id or version_name here, see uploader/
      store.rb for details.
  # def filename
  #   "something.jpg" if original_filename
  # end

end
```

A.2.2 Serializers

User serializer

```
class UserSerializer < ApplicationSerializer
  attributes :id, :username, :email, :birthdate, :
      description, :gender, :followings_count, :
      followers_count, :posts

  def posts
    object.posts.pluck(:id)
  end
end
```

User follow serializer

```
class UserFollowSerializer < ApplicationSerializer
  attributes :id, :username
end
```

Post serializer

```
class PostSerializer < ApplicationSerializer
  attributes :id, :image, :description, :author

  def image
    "/uploads/post/image/#{object.id}/file.png"
  end

  def author
    object.author_id
  end
end
```

Like serializer

```
class LikeSerializer < ApplicationSerializer
  attributes :id, :user

  def user
    object.user_id
  end
end
```

Comment serializer

```
class CommentSerializer < ApplicationSerializer
  attributes :id, :comment, :author, :post

  def author
    object.author_id
  end

  def post
    object.post_id
  end
end
```

A.2.3 Controllers

Users controller

```
class UsersController < ApplicationController
  include Pagingable
  #before_filter :restrict_access, except: [:sign_up, :
     log_in, :feed]
  def sign_up
    user = User.new(signup_params)
    if user.save
      response = {success: true, result: user.id}
      render json: response, status: :created
    else
      response = {success: false}
      render json: response, status: :internal_server_error
    end
  end

  def log_in
    params =  login_params
    token = User.authenticate(params['username'], params['
       password'])
    if token
      response = {success: true, result: token}
      render json: response, status: :ok
    else
      response = {success: false}
      render json: response, status: :unauthorized
    end
  end

  def show
    user = User.find(params.require(:id))
    if stale? user
      response = { success: true, result: UserSerializer.new
         (user, root: false) }
      render json: response , status: :ok
    end
  end

  def index
    offset, limit = pagination_values
    users = User.select(:id, :username, :email, :birthdate,
       :description, :gender, :followings_count, :
       followers_count, :updated_at).order(:id).offset(
       offset).limit(limit)
    if stale? users
```

```
    render json: UserSerializer.array_to_json(users, {
      success: true, offset: offset, limit: limit}),
      status: :ok
  end
end

def following
  offset, limit = pagination_values
  user = User.find(params.require(:id))
  followings = user.followings.offset(offset).limit(limit)
  if stale? followings
    render json: UserSerializer.array_to_json(followings,
      {success: true, offset: offset, limit: limit}),
      status: :ok
  end
end

def followers
  offset, limit = pagination_values
  user = User.find(params.require(:id))
  followers = user.followers.offset(offset).limit(limit)
  if stale? followers
    render json: UserSerializer.array_to_json(followers, {
      success: true, offset: offset, limit: limit}),
      status: :ok
  end
end

def following_posts
  offset, limit = pagination_values
  following_ids = UserFollowing.where(user_id: id).pluck(:
    following_id)
  posts = Post.select(:id, :description, :author_id, :
    created_at, :updated_at).where(author: following_ids)
    .order(created_at: :desc).offset(offset).limit(limit)
  if stale? posts
    render json: PostSerializer.array_to_json(posts, {
      success: true, offset: offset, limit: limit}),
      status: :ok
  end
end

def followers_posts
  offset, limit = pagination_values
```

```
    id = params.require(:id)
    follower_ids = UserFollowing.where(following_id: id).
      pluck(:user_id)
    follower_posts = Post.select(:id, :description, :
      author_id, :created_at, :updated_at).where(author:
      follower_ids).order(created_at: :desc).offset(offset)
      .limit(limit)
    if stale? follower_posts
      render json: PostSerializer.array_to_json(
        followers_posts, {success: true, offset: offset,
        limit: limit}) , status: :ok
    end
  end

  # A user's feed is all posts made by that user or any of
    the users it follows
  # ordered by time of posting
  def feed
    offset, limit = pagination_values
    id = params.require(:id)
    feed_users_ids = UserFollowing.where(user_id: id).pluck
      (:following_id) << id
    feed_posts = Post.select(:id, :description, :author_id,
      :created_at, :updated_at).where(author:
      feed_users_ids).order(created_at: :desc).offset(
      offset).limit(limit)
    if stale? feed_posts
      render json: PostSerializer.array_to_json(feed_posts,
        {success: true, offset: offset, limit: limit}) ,
        status: :ok
    end
  end

  private
  def login_params
    params.permit(:username, :password)
  end

  def signup_params
    params.permit(:username, :email, :password, :birthdate,
      :description, :gender)
  end
end
```

Posts controller

```
class PostsController < ApplicationController
  include TagSearchable
  before_filter :restrict_access, except: :show

  def show
    post = Post.find(params.require(:id))
    if stale? post
      response = {
        success: true,
        result: PostSerializer.new(post, root: false)
      }
      render json: response
    end
  end

  def create
    post = Post.new
    post.author = @authenticated_user
    post.description = params[:description]
    post.image = params[:image]
    tags = find_tags(params[:tags])
    user_tags = find_user_tags(params[:user_tags])
    id = post.create_assoc_and_save(tags, user_tags)
    if id
      render json: {success: true, result: id}, status: :
        created
    else
      render json: {success: false}, status: :
        internal_server_error
    end
  end

  def update
    post = Post.find(params.require(:id))
    post.description = params[:description] if params.
      has_key?(:description)
    post.tags = find_tags(params[:tags]) if params.has_key
      ?(:tags)
    post.tagged_users = find_user_tags(params[:user_tags])
      if params.has_key?(:user_tags)
```

```
    if post.save
      render json: {success: true}, status: :ok
    else
      render json: {success: false}, status: :
          internal_server_error
    end
  end

end
```

Likes controller

```
class LikesController < ApplicationController
  before_filter :restrict_access, except: :post_likes

  def create
    post = Post.find(params.require(:id))
    Like.like(@authenticated_user, post)
    render json: {success: true}, status: :created
  end

  def post_likes
    post = Post.find(params.require(:id))
    post_likes = post.likes
    if stale? post_likes
      render json: LikeSerializer.array_to_json(post_likes,
          {success: true}) , status: :ok
    end
  end
end
```

Comments controller

```
class CommentsController < ApplicationController
  include TagSearchable
  include Pagingable
  before_filter :restrict_access, except: [:show, :
      post_comments]

  def show
    comment = Comment.find(params.require(:id))
    if stale? comment
      response = {
        success: true,
        result: CommentSerializer.new(comment, root: false)
```

```
    }
    render json: response, status: :ok
  end
end

def create
  comment = Comment.new
  comment.comment = params.require(:comment)
  comment.post = Post.find(params.require(:post))
  comment.author = @authenticated_user
  tags = find_tags(params[:tags])
  user_tags = find_user_tags(params[:user_tags])

  id =  comment.create_assoc_and_save(tags, user_tags)
  if id
    render json: { success: true, result: id }, status: :
       created
  else
    render json: {success: false}, status: :
       internal_server_error
  end
end

def update
  comment = Comment.find(params.require(:id))
  comment.comment = params[:comment] if params.has_key? :
     comment
  comment.tags = find_tags(params[:tags]) if params.
     has_key?(:tags)
  comment.tagged_users = find_user_tags(params[:user_tags
     ]) if params.has_key?(:user_tags)

  if comment.save
    render json: { success: true }, status: :ok
  else
    render json: { success: false }, status: :
       internal_server_error
  end
end

def post_comments
  offset, limit = pagination_values
  comments = Comment.select(:id, :comment, :author_id, :
     post_id, :updated_at).where(post_id: params.require(:
```

```
        id)).offset(offset).limit(limit)
    if stale? comments
      render json: CommentSerializer.array_to_json(comments,
          {success: true, offset: offset, limit: limit}),
        status: :ok
    end
  end
end
```

Concerns

```
module Pagingable
  extend ActiveSupport::Concern
  private
  def pagination_values
    offset = (params[:offset] || 0).to_i
    # Set limit to supplied value if any otherwise default
       to 10
    # max limit 100 per request
    limit =  (params[:limit] || 10).to_i
    limit = 100 if limit > 100
    return offset, limit
  end
end

module TagSearchable
  extend ActiveSupport::Concern
  private
  def find_tags(tags_param_arr)
    tags = []
    if(tags_param_arr)
      tags_param_arr.each do |tag|
        tags << Tag.find_or_create_by(text: tag)
      end
    end
    tags
  end

  def find_user_tags(user_tags_param_arr)
    # Find all users that have been tagged
    user_tags = []
    if(user_tags_param_arr)
      user_tags_param_arr.each do |user_tag|
        user = User.find_by(username: user_tag)
        user_tags << user if user
```

```
      end
    end
    user_tags
  end
end
```

A.2.4 Tasks

DB generation rake task

```
namespace :db do
  desc "Clears and sets correct values for counter caches"
  task set_counter_cache: :environment do
    User.find_each do |u|
      User.reset_counters(u.id, :followers)
      User.reset_counters(u.id, :followings)
    end
  end
  desc "Creates a dataset to use for testing, takes size as
     an environment variable"
  task generate: :environment do
    size = ENV['size'] || 100
    size = size.to_i
    start_time = Time.now

    # Generate tags
    tags = []
    size.times do
      tags << FactoryGirl.create(:tag)
    end

    size.times do |i|
      if i % 10 == 0
        puts "#{i/size} % done (#{i} of #{size}) in #{(Time.
            now - start_time).round(4)} seconds"
      end
      user = FactoryGirl.create(:user)

      # Each users has 0 - 50 posts
      rand(0..50).times do
        # Create 0 - 5 tags and user_tags for each post
        user_tags = []
        post_tags = []
        rand(0..5).times do
          user_tags << User.all.sample
```

```
            post_tags << tags.sample
          end

          FactoryGirl.create(:post, author: user, tags:
              post_tags, tagged_users: user_tags)
        end

        # Each user makes 0 - 100 comments
        rand(0..100).times do
          FactoryGirl.create(:comment, author: user, post_id:
              Post.all.sample)
        end

        # Each user likes 0 - 200 posts
        rand(0..200).times do
          Like.like(user, Post.all.sample)
        end

        # Each user folows 0 - 100 other users
        rand(0..100).times do
          user.follow(User.all.sample)
        end
      end
      puts "Data generation completed in #{(Time.now -
          start_time).round(4)}"
    end
end
```

JMeter test plan generation task

```
##
# Task for generating JMeter test plans for a set of API
    endpoints that are used for benchmarking the API
# either run with default settings 25 threads for small and
    100 threads for large
# or number of threads can be overridden by specifying a
    threads parameter
#
namespace :benchmark do
  port = 3000
  desc "Benchmarks /users with 25 concurrent threads for 120
      s"
  task users: :environment do
    thread_count = num_threads
```

```
  ids = setup
  test do
    threads count: thread_count, rampup: 5, duration: 120
       do
      header({name: 'Authorization', value: "Token #{ids[:
         user].token}"})
      visit name: '/users', url: "http://localhost/users",
      port: port
    end
  end.jmx(file: "benchmark/users_#{thread_count}_testplan.
     jmx")
end

desc "Benchmarks /user/:id/feed with 25 concurrent threads
    for 120s"
task feed: :environment do
  thread_count = num_threads
  ids = setup
  test do
    threads count: thread_count, rampup: 5, duration: 120
       do
      header({name: 'Authorization', value: "Token #{ids[:
         user].token}"})
      visit name: '/user/:id/feed', url: "http://localhost
         /user/#{ids[:user].id}/feed",
      port: port
    end
  end.jmx(file: "benchmark/feed_#{thread_count}_testplan.
     jmx")
end

desc "Benchmarks /user/:id/followers with 25 concurrent
   threads for 120s"
task followers: :environment do
  thread_count = num_threads
  ids = setup
  test do
    threads count: thread_count, rampup: 5, duration: 120
       do
      header({name: 'Authorization', value: "Token #{ids[:
         user].token}"})
      visit name: '/user/:id/followers', url: "http://
         localhost/user/#{ids[:user].id}/followers",
      port: port
```

```
      end
    end.jmx(file: "benchmark/followers_#{thread_count}
       _testplan.jmx")
  end

  desc "Benchmarks /post/:id/comments with 25 concurrent
     threads for 120s"
  task post_comments: :environment do
    thread_count = num_threads
    ids = setup
    test do
      threads count: thread_count, rampup: 5, duration: 120
         do
        header({name: 'Authorization', value: "Token #{ids[:
           user].token}"})
        visit name: '/post/:id/comments', url: "http://
           localhost/post/#{ids[:post].id}/comments",
        port: port
      end
    end.jmx(file: "benchmark/post_comments_#{thread_count}
       _testplan.jmx")
  end

  task all: :environment do
    thread_count = num_threads
    ids = setup
    test do
      threads count: thread_count, rampup: 5, duration: 120
         do
        header({name: 'Authorization', value: "Token #{ids[:
           user].token}"})
        visit name: '/users', url: "http://localhost/users"
        visit name: '/user/:id/feed', url: "http://localhost
           /user/#{ids[:user].id}/feed"
        visit name: '/user/:id/followers', url: "http://
           localhost/user/#{ids[:user].id}/followers"
        visit name: '/post/:id/comments', url: "http://
           localhost/post/#{ids[:post].id}/comments",
        port: port
      end
    end.jmx(file: "benchmark/all_#{thread_count}_testplan.
       jmx")
  end
```

```
  desc "Delete all files in benchmark folder"
  task clear: :environment do
    FileUtils.rm_rf 'benchmark/.'
  end

  private
  # Either set the number of threads manually or default to
     25
  def num_threads
    begin
      return Integer(ENV['threads'])
    rescue
      return 25 # Default to small with no args
    end
  end

  def setup
    User.connection
    Post.connection
    ids = {}
    # Get the user who follows the most users to make feed
       as heavy as possible for a worst case scenario
    ids[:user] = User.select('users.*, COUNT(user_followings
       .id) AS followings_count').joins(:user_followings).
       group('users.id').order('followings_count DESC').
       limit(1).first
    # Get the post with the moste comments to make post
       comments as heavey as possible for a worst case
       scenario
    ids[:post] = Post.select('posts.*, COUNT(comments.id) AS
        comments_count').joins(:comments).group('posts.id').
       order('comments_count DESC').limit(1).first
    User.authenticate(ids[:user].username, "password")
    ids[:user].reload
    return ids
  end
end
```