FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Scaling Rails: a system-wide approach to performance optimization

Gonçalo Santarém da Silva

Master in Informatics and Computing Engineering

Supervisor: Ademar Manuel Teixeira de Aguiar (PhD.)

28^{th} June, 2010

Scaling Rails: a system-wide approach to performance optimization

Gonçalo Santarém da Silva

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: João Manuel Paiva Cardoso (Associate Professor)

External Examiner: Alberto Manuel Rodrigues da Silva (Associate Professor)

Internal Examiner: Ademar Manuel Teixeira de Aguiar (Auxiliary Professor)

21st July, 2010

Abstract

Web's popularity and importance on everyday life increases day by day. Its users have high expectations and require better user experiences in their daily interactions. Ruby on Rails was created as a tool to help coping with this demand. This framework is criticized for issues with scalability. The lack of general guidelines, profiling tools, and global awareness of the importance of building highly performant applications all contribute to this problem.

Producing the aforementioned guidelines, greatly improving Rails-related profiling tools and increasing this subjects' awareness inside the community are the main goals of this project.

System components have been addressed in three phases: benchmark, tweak and develop, all from a Rails-centered perspective. Benchmarking operating systems involved comparing various Linux distributions and FreeBSD. Gentoo was the best performing alternative. As of tweaking, kernel configurations were explored. A generic benchmarking script for UNIX systems was developed.

Regarding Ruby interpreters, benchmarks showed that YARV outperforms and improves the scalability of Rails applications over MRI. Its garbage collector's flexibility was enhanced for adaptive performance. Its storage and retrieval capabilities of profiling information were extended and a graphical profiling output integrated.

As of web servers, a memory usage monitoring script was developed. All web servers yielded similar performance results, although Thin had a remarkably lower memory consumption. Nginx was the best performing reverse proxy. Finally, many configuration options were explored.

Regarding databases, a Ruby library for MySQL was improved.

Concerning Rails, common pitfalls were exposed and solutions presented, exploiting their performance differences. Its profiling tools were improved and now seamlessly integrate with Ruby's. Redmine and many plugins were ported to the latest versions of Ruby and Rails. Finally, an article series on performance optimization was started, as well as the development of an official benchmarking suite for performance-oriented continuous integration.

The aforementioned general guidelines were created. The native profiling tools of Ruby and Rails were refactored. Revamping these tools, upgrading Redmine and plugins, publishing an article series, adding flexibility to YARV's garbage collector and developing an official benchmarking suite are activities which generally contribute to the awareness of this subjects' importance.

Resumo

A importância da Web tem aumentado progressivamente. Os utilizadores têm expectativas elevadas, exigindo experiências de alta qualidade. O Ruby on Rails ajuda a colmatar estas exigências, embora se critique a sua escalabilidade. A falta de guias gerais, ferramentas de análise de desempenho e desconhecimento da importância do mesmo contribuem para este problema.

Os objectivos principais deste trabalho consistem em criar os referidos guias genéricos, melhorar as ferramentas de análise de desempenho existentes e aumentar a importância desta temática.

Trabalharam-se os componentes em três fases: testes, configurações e desenvolvimento, sempre da perspectiva do Rails. Quanto aos sistemas operativos, compararam-se várias distribuições de Linux e o FreeBSD. O Gentoo foi a alternativa com melhor desempenho. Várias opções do *kernel* foram exploradas. Foi desenvolvido um *script* de teste genérico de desempenho para sistemas UNIX.

Quanto ao Ruby, o YARV mostrou ter melhor desempenho que o MRI. Várias opções de parametrização foram introduzidas no seu *garbage collector*. A sua capacidade de análise de desempenho foi, ainda, melhorada.

No que toca aos servidores web, foi desenvolvido um *script* de monitorização de memória. Todos eles evidenciam um desempenho similar, embora o Thin se destaque pela baixa utilização de memória. O Nginx obteve os melhores resultados como *reverse proxy*.

Relativamente às bases de dados, melhorou-se uma biblioteca Ruby para MySQL.

Quanto ao Rails, vários problemas recorrentes foram analisados e foram propostas soluções. As ferramentas nativas de análise de desempenho foram melhoradas e agora integram com as do Ruby. A Redmine e vários *plugins* foram portados para as últimas versões do Ruby e do Rails. Por fim, deu-se início a uma série de artigos sobre optimização de desempenho, bem como, se iniciou o desenvolvimento de uma bancada oficial de testes de desempenho para a integração contínua do Rails.

Assim, foram criadas linhas de guia gerais e aprimoraram-se as ferramentas de análise de desempenho do Ruby e do Rails. Estas melhorias, a actualização da Redmine e *plugins*, a série de publicações, o aumento de flexibilidade do YARV e a melhoria da integração contínua do Rails são actividades que ajudam a aumentar a importância geral deste assunto na comunidade.

Acknowledgements

To everyone at *Escolinhas.pt*, for allowing me to mix work and fun on a daily basis.

To Ademar Aguiar and Nuno Baldaia, for their patience at guiding a sometimes slugabed student.

To Muhammed Ali, for helping me set my priorities when coding tons of C was what I really wanted to do.

To Brian Lopez, for happily sharing his knowledge on Ruby C extensions when all the documentation I could find was written in Japanese.

To Rupak Ganguly, for helping me polish and publish my first magazine article ever.

To Yehuda Katz, for guiding me through the Ruby Summer of Code and trusting me every single time, even when it meant breaking Rails.

To Pedro Coelho, for lending me his computer when all official means have failed.

To Paulo Pereira, for revising this report, pointing me in the right direction when I started and for continuously motivating me.

To everyone who stained their black cloaks with me, for teaching me things I could not have learned elsewhere.

To Sara, for all her love, help and support, for cheering me up whenever I felt down and for making me feel like I am the best person in the world.

To my family, for their patience and support while I pulled dozens of all nighters during this awesome course, and for giving me the possibility to make the most out of it.

“Fast isn‘t a feature. Fast is a requirement.”

— Jesse Robbins

Contents

List of Figures

LIST OF FIGURES

List of Tables

Definitions

The following definitions are in alphabetical order.

Ahead-of-time Compilation Compilation of intermediate code into a system-dependent binary.

Continuations C-like gotos for Ruby.

Coupling The degree to which each program module relies on each one of the other modules.

Endian Order of individually addressable sub-units within a longer data word in external memory.

EventMachine A library for Ruby, C++ and Java that provides event-driven I/O using the Reactor pattern.

Fork A processes' act of creating a copy of itself.

Green Threads Threads scheduled by the Virtual Machine, emulating multi-threaded environments.

Just-in-time Compilation Also known as dynamic translation, just-in-time compilation is the act of converting code at runtime prior to executing it natively.

Mark and Sweep The first garbage collection algorithm, consisting of two blocking phases: a mark phase and a sweep phase.

RDoc The embedded documentation generator for the Ruby programming language.

World Wide Web System of interlinked hypertext documents contained on the Internet.

Abbreviations

The following abbreviations are in alphabetical order.

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CERN *Conseil Européen pour la Recherche Nucléaire*, currently known as European Organization for Nuclear Research

CFQ Completely Fair Queuing

CPU Central Processing Unit

CSV Comma-Separated Values

DBMS Database Management System

DRY Don't Repeat Yourself

ERB Embedded Ruby

HTTP Hypertext Transport Protocol

IP Internet Protocol

I/O Input/Output

JVM Java VM

MB Megabyte

MRI Matz's Ruby Interpreter

MVC Model-View-Controller

MPM Multi-Processing Module

ORM Object-Relational Mapping

OS Operating System

PID Process ID

PHP PHP Hypertext Preprocessor

POSIX **P**ortable **O**perating **S**ystem **I**nterface [for Unix]

Rails Ruby on Rails

RDBMS Relational DBMS

REE Ruby Enterprise Edition

STL Standard Template Library

TCP Transmission Control Protocol

UNIX Uniplexed Information and Computing Service

USA United States of America

VCS Version Control Systems

VM Virtual Machine

Web World Wide Web

YARV Yet Another Ruby VM

.NET Microsoft .NET Framework

Chapter 1

Introduction

This chapter briefly presents the project's context, purpose and scope. It covers the motivation behind it and its objectives.

1.1 Context

The internet started in the early 1990s as a work tool for CERN. It evolved into a vast information repository for the public [WZ09]—the Web—and had 16 million users back in 1995. The year after Netscape went public and achieved an impressive 90% market share. It quickly lost its prominence during the first browser war, conceding its leadership to Microsoft's Internet Explorer [WWL05]. The Web was starting to become a presence in everyday life. In 2009, 14 years later, the Web had more than 1700 million users and its popularity keeps growing nowadays [Gro].

The Web is becoming an essential pillar of many businesses, social networks, gaming industries *et al.* since distance is no longer an issue when it comes to information exchange. It begins to present itself as a critical presence on computers nowadays. Many people believe future applications will mostly be web-based, pushing internet's importance even further [TMIP08].

The growth of internet usage and its impact in corporate applications implies all kinds of particularities. First of all, users must trust the web. This is an essential pillar of any web service. To build customer trust, service providers must pay attention to their users' needs and desires and they must meet their expectations [AG07].

In the fall of 2001, many people concluded that the web was overhyped and bloated. There was a need for richer content and better user experiences [O'R05]. In this context, the term *Web 2.0* was born. Its meaning is not well established but a commonly accepted definition states that it consists in the improvement of the first version of the Web [Vis08] and involves many core concepts like usability and dynamic content [Lew06].

Side by side with the Web's growth were the increasing user expectations when it comes to their experiences. As time goes by, users demand better interactions with the services they use.

Their user experiences are partially based on response times, responsiveness and performance of Web applications [NL06]. These concepts are part of the key factors to their success [RYS09]. Users expect waiting times to be kept at an acceptable minimum and whenever they feel that this expectation is not being met their trust on the service diminishes. When users enter a given website they have a limited patience, related to their expectations and previous experiences. Whenever the system fails to meet those expectations, their patience decreases and it can cause them to leave. Steve Krug[1] entitles this phenomenon as the *Reservoir of Goodwill* [Kru05].

This increasing popularity, dependence and demand on the Web strives the developers for better tools to build quality web applications. Most developers seek the ability to increase their productivity while being able to build more complex, full-featured systems that suit their users' needs, either it is a social or business-oriented service [IFVDCI08].

With the growing importance of Web applications, many tools emerged trying to make the developer's life easier. *Web 2.0* improved the internet and created a new set of needs and expectations. This need motivated the development of powerful frameworks and, among many others, the Ruby on Rails framework[2] was born [Lew06]. As the community puts it [Hana]:

> "Ruby on Rails® is an open-source web framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration."

Ruby on Rails is one of the best-known Ruby frameworks [THB+06]. Most people, from individuals to companies, want a *Web 2.0* killer application and Rails seems an excellent way of achieving it [Got05]. This framework is commonly associated with the *Web 2.0* concept, along with AJAX [MT08]. It is also deeply related with *Agile Web Development* [THB+06]. Rails allows developers to build high quality applications with smaller effort—less time, less lines of code and less files, always with low coupling [SJW08].

As Tim O'Reilly[3] states [O'R]:

> "Ruby on Rails is a breakthrough in lowering the barriers of entry to programming. Powerful web applications that formerly might have taken weeks or months to develop can be produced in a matter of days."

The huge success achieved by this framework made it jump into a spotlight, with many companies starting to use it as the development framework for their applications. Some widely known services like *Basecamp*[4], *Twitter*[5], *Hulu*[6], *YellowPages*[7] or *GitHub*[8] push this platform even further by giving it even more popularity [Hanb].

[1] Steve Krug is the author of the renowned book about human computer interaction and web usability entitled *Don't Make Me Think*

[2] http://rubyonrails.org

[3] Tim O'Reilly is the founder of O'Reilly Media and a supporter of the free software and open source movements

[4] http://basecamphq.com/

[5] http://twitter.com/

[6] http://hulu.com/

[7] http://yellowpages.com/

[8] http://github.com/

Basecamp, the original Rails application [Hanb], is an online project management tool which features live collaboration and task aiding software. It gathered over 1 million users since 2006 [37s]. It is ranked 516^{th} on the *Alexa Traffic Rank* [AI].

Twitter, a real-time short messaging service that works on multiple networks and devices. It is used for quick sharing of information, either updates from friends or breaking world news updates. It had more than 18 million adult users in the USA by the end of 2009 and is expected to achieve an impressive quantity of 26 million adult users in the same country by 2010 [eMa]. It is ranked 11^{th} on the *Alexa Traffic Rank* [AI].

Hulu, a TV and Movie streaming website which allows users to watch their favorite shows on the browser for free. It had a significant amount of traffic in 2009, only staying behind Google services and *Fox Interactive Media* [com09]. It ranks 177^{th} on the *Alexa Traffic Rank* [AI].

YellowPages, a service that indexes and provides business listings of the United States of America, allowing users to search for services they are looking for, among other features. It ranks 929^{th} on the *Alexa Traffic Rank*. To note that it is a USA-oriented application, raking 161^{th} if the scope is limited to the USA [AI].

GitHub, one of the best know repository hosting system which works with the Git VCS. It currently has more than 185 thousand registered developers and although being tightly associated with public open-source projects, it also supports private development. It ranks 997^{th} on the *Alexa Traffic Rank* [AI].

These systems have high scalability demands. In order to keep increasing their popularity and to keep building users' trust, they also need to provide a great user experience which involve reasonable response times while serving thousands of requests concurrently.

Some press reports question Rails' ability to scale, mainly based on the issues *Twitter* faced when its growth reached a given magnitude [Ken07]. However, most of the issues were demystified as a software architecture design issue, taking the blame off of Ruby on Rails [Inf09]. Nonetheless, despite all the advantages this framework possesses, scalability is not one of them. Although not being as scalable as PHP or Java, Ruby offers higher development speeds [TB06].

Luckily, only high traffic websites have to get deeply involved in scalability details. However, developers should always aim at building highly performant applications from the start. They should be able to build Rails-based high-quality applications whose scalability is not directly related to hardware upgrades [Ken07]. Issues should be identified and solutions proposed so that this acclaimed Ruby framework becomes more scalable *out of the box*, diminishing its dependence on hardware upgrades or major architectural changes. Developers should be aware of their choices' benefits and shortcomings. This way, Ruby on Rails development happiness can not only last through the creation of a web platform, but also through its maintenance.

1.2 Motivation and Objectives

The Web starts to play a critically important role in many people's lives, either from a professional or personal point of view. User experiences have become of great importance in recent times, with *Web 2.0* raising the expectations on better interactions.

Internet accesses keep increasing in number and the recent developments in the smartphone, tablet and netbook's areas help inflating this growth rate even further. More people are starting to be permanently connected to the internet [NPD09, IT 09, USA09].

Users expect the Web to work as they preconceived and this fact has a great focus on recent developments in Interaction Design [PRS01]. As innovation pushes user experiences to a new level, with richer information presented and organized in ways never seen before, technologies tend to emerge to support such evolutionary content and forms of organization. Developers need to meet the users' needs and they do not have unlimited time to do it, thus many recent frameworks have gained notorious popularity for being agile and robust, increasing productivity rates to a higher level [Jaz07]. Ruby on Rails, as most recent frameworks, offers convenient methods and features which greatly improve the product's quality without the need for extended development times. However, it also makes it harder for the development team to build a highly scalable application when there are limited hardware resources [Kae06].

Scaling and performance optimization should not be so hard to achieve in Rails, though. Many *Web 2.0* platforms are created everyday and Rails-related scalability issues should not be an obstacle to their success. The framework's purpose is to help developing high-quality applications, not limiting their accomplishments to a given number of concurrent users. The Web should be able to shine in all of its glory and tools like Ruby on Rails exist to allow it to improve and innovate further and further, meeting the users' increasing standards and demands.

Performance optimization has been a work focus since computers were born [DZ65]. Many people have focused on optimizing many different components, like the Ruby interpreter [Sas05], Rails itself [Kat09b] or the superjacent application [Hof06, Kat09a, Sla08, EY09]. Many have focused on improving the speed and scalability of databases [HF86] and webservers [SHB06]. Others look for potential issues in the Operating System and end up patching and tweaking the system's configuration [LB96, RBH+95]. One can infer that most of the performance optimization activities focus on single elements or try to find out a single culprit to blame. It is also necessary to envision performance optimization as a holistic activity. If a small part of the system changes it will affect all those who interact with it by smaller or larger margins. In Chaos Theory this is called *Sensitive Dependence* or, as more commonly known, the *Butterfly Effect*. As mentioned in *Quantum Chaotic Environments* [KJZ02]:

> "The exponential divergence (...) from slightly different initial conditions—the famous butterfly effect—is a fingerprint of chaos in classical mechanics."

Rails is a highly dependent system. There are many components involved and all of them can be optimized. The key concept is that in order to improve a Rails application's scalability, the task should be addressed with a deep notion of its associated components and the available alternatives.

The whole system performance is what truly matters, not the performance of its individual parts. The core mindset of this project is to address all involved components from Rails' perspective.

There is the need for a solid set of general development conventions and guidelines oriented towards the scalability and performance of Rails projects. Developers seek optimal configurations for all the components involved so they can wring every processing cycle out of their applications, in order to increase their scalability and decrease their response times. There is urgency in looking at all components Rails depends on, determining which are best for which situation and tune them to suit Rails' needs—the system, envisioned as a whole.

Improving user experiences while profiling applications is also imperative. Profiling is a critical activity when optimizing applications but recent versions of Ruby and Rails break these tools and older versions only supported text-based outputs. Profiling should be functional, less verbose and more intuitive.

Finally, increasing the global awareness of the importance of building highly performant applications is also important. Developers should consider this aspect from the beginning and have access to all the information they need to plan, architecture and develop scalable systems.

These activities will be mostly applied in *Escolinhas.pt*[9], a rapidly growing Portuguese Rails-based project. *Escolinhas.pt* aims at sustaining social and collaborative work for children in elementary schools involving students, teachers and parents as its users. With the user demand increasing day by day, it becomes an excellent case-study application to research, test and apply most of the work and discoveries made during the course of this thesis.

Producing the aforementioned generic conventions and guidelines, fixing and greatly improving Rails-related profiling tools and increasing this subjects' awareness inside the Ruby on Rails community are the main goals of "Scaling Rails: a system-wide approach to performance optimization".

1.3 Report Overview

The rest of this report is structured as follows.

Chapter 2: "Technologies Review" gives an overview over the involved technologies mentioned in this research, providing important background information on each one of them.

Chapter 3: "State of the Art" reviews each component alternative's performance and scalability and provides other important details for the problem approach.

Chapter 4: "Problem Statement" contains the problem to be addressed and throughly explains it, also exposing its usefulness.

Chapter 5: "Problem Approach and Results" benchmarks each component's alternatives, analyzes their settings and possible configurations, and details the developments made concerning each component.

[9] http://escolinhas.pt

Chapter 6: "Conclusions" reviews the project, drawing conclusions about the issues addressed in 4. It also provides a summary of contributions and some insights on which future developments have been considered.

Chapter 2

Technologies Review

Since this research spans into multiple traditional fields, this chapter provides an overview of the technologies involved. A high-level analysis is made concerning their most important characteristics and a solid knowledge base is provided as a sustaining base for exposing their state of the art, further analysis and development.

2.1 Operating Systems

The operating system is the base of everything else. It runs on top of the hardware and allows applications to use its resources. There are many kinds of operating systems, with different characteristics and philosophies. The relevant ones to this research will be explained in the following sections.

2.1.1 GNU/Linux

Linux is the term used to describe UNIX-based operating systems that run the Linux *Kernel*. It was created in 1991 by Linus Torvalds [Kor06], who is also the author of *git* [Cha09]. It is one of the most significant open-source projects where volunteers from all over the world work together to achieve a common goal—improve the operating system itself [Kor06]. While having low usage on desktop systems [AWS], Linux is widely used on servers, mainframes and super computers. It is commonly known for its security and reliability. One sustaining example is the list of operating systems used by the most reliable hosting companies in December 2009, where Linux figures 6 times in the top 10 [Net].

Linux stands as the base for many UNIX-like software distributions, specifically Linux distributions. These consist on a large collection of software applications and configurations which range from full-featured desktop systems to minimal environments. Non-commercial Linux distributions commonly used in server environments include:

Debian, maintained by a developer community with a strong commitment to free software principles[1].

Ubuntu Server, derived from Debian and maintained by Canonical, being the server version of the most popular Linux distribution[2].

CentOS, derived from the same sources used by the renowned *Red Hat*[3] distribution.

Gentoo, known for its FreeBSD Ports-like system for custom compiling[4] of applications.

Each distribution aims at adding its own flavor to the Linux operating system, providing different user experiences to their users [Mor02].

2.1.2 Berkeley Software Distribution

Berkeley Software Distribution, also know as BSD or *Berkeley Unix*, is considered to be a branch of the *Unix* operating system. It was created in 1977 in the University of California in Berkley by the Computer Systems Research Group. Entitled by some as the greatest software ever written [Bab06], even Linux's creator, Linus Torvalds himself, went as far as stating [Lin93]:

> " If 386BSD had been available when I started on Linux, Linux would probably never had happened"

Just like Linux, it is open-source software and has several associated distributions, although at a smaller scale. FreeBSD[5] is the most popular BSD distribution [Auz09].

A significant remark is that both Apple's *Mac OS X* and Microsoft's *Windows* use parts of FreeBSD's source code [App, eve01].

2.1.3 Microsoft Windows Server

Windows Server is Microsoft Corporation's operating system oriented towards servers. The current version is entitled *Windows Server 2008* and, as its name implies, was released in 2008. It is built on top of the same code base used in *Windows Vista*.

Windows is proprietary software and consequently does not come in a distribution manner like the *UNIX*-based operating systems mentioned before.

2.1.4 Mac OS X Server

Mac OS X Server is Apple's server-oriented operating system. It is architecturally identical to its desktop counterpart, except that it includes work group management and administration software tools.

[1] http://www.debian.org/
[2] http://www.ubuntu.com/products/whatIsubuntu/serveredition
[3] http://www.centos.org/
[4] http://www.gentoo.org/
[5] http://www.freebsd.org/

This operating system is usually found on rack mounted server computers which are also designed by Apple.

2.2 Ruby

Ruby is a dynamic and object-oriented language created by Yukihiro Matsumoto, who released it to the public in 1995. The purpose was to create a "language that was more powerful than Perl, and more object-oriented than Python" [Ste01].

Ruby was inspired by languages such as Lisp, Smalltalk and Perl, and its core characteristics and features include [Rub, FM08]:

Open source. Ruby's license allows anyone to use, copy, modify or distribute it.

Pure object-oriented. In Ruby, everything is an object, including classes, modules and data types—even numbers, booleans and null values which in other object-oriented languages are known as "primitives". An object's properties are known as "instance variables" and actions are "methods".

Flexible. It not only features dynamic typing, but also very powerful reflective and metaprogramming capabilities. Ruby does not restrict what a programmer can do. Any part of Ruby code can be removed, redefined or extended, even at runtime. This is not only true for a programmer's own code, but also for core Ruby classes such as Object and String.

Automatic memory management. Like other languages such as Java and contrary to C, developers do not manage the program's memory usage.

Portable. Ruby is mainly developed in GNU/Linux but can run on most operating systems and platforms such as BSD, Mac OS X, Windows 95 and many others.

Exception handling. Ruby can recover from errors just like Java, C++ and Python.

Ruby started to become popular in 2001, with the start of the *RubyGems*[6] project, which allows easy packaging and distribution of applications and libraries [Wil07]. Ruby has two main versions, 1.8 and 1.9, the last one having been publicly released approximately two years ago [Son09].

2.2.1 Ruby 1.8

The most commonly found version of Ruby in Rails' projects is the 1.8, mainly due to the fact that it was the officially recommended version for many years.

[6] http://rubygems.org/

2.2.1.1 MRI

Ruby's 1.8 official interpreter was developed by the language's creator—Yukihiro Matsumoto, also known as *Matz*—and its first public release happened in 1995. Some people mistake MRI for "Main Ruby Implementation" but the abbreviation is actually related to its creator's name—*Matz Ruby Interpreter*. Some particular characteristics of this interpreter include:

Language. MRI is written in C.

Threading. It can emulate a threaded environment without relying on the operating system capabilities by using *green threads*.

Garbage Collector. The garbage collector is based on the simple *mark-and-sweep* algorithm.

Extensions. Developers can extend Ruby's basic functionality by writing their own extensions. These can be written in Ruby or in native C, by using MRI's powerful API.

Bytecode Interpretation. MRI lacks a bytecode interpreter. When a program is executed, it parses its source and creates its syntax tree. Then, it iterates over this tree directly while executing the program.

MRI was the only official Ruby interpreter for many years and, consequently, it is the most widely used.

2.2.1.2 Ruby Enterprise Edition

This Ruby interpreter, called REE, is based on MRI's source code. However, it includes many Rails-oriented enhancements. It was first released in 2003 and has been merging with MRI periodically, keeping its own changes aside. Its characteristics and differences from the official interpreter for version 1.8—MRI—include [Phub]:

Language. Since REE is based on MRI, it is written in C.

Threading. The thread implementation is the same found on MRI.

Garbage Collector. The garbage collector was improved, being *copy-on-write* friendly. This allows for reduced memory usage when paired with Passenger, who uses *preforks* in combination with this feature. It also enables the user to tweak its settings for adaptive performance.

Extensions. Just like MRI, it natively supports Ruby and C extensions.

Bytecode Interpretation. REE is mostly based in MRI's source code, as mentioned before, so it lacks a bytecode interpreter. It also iterates over the program's syntax tree directly.

Other Differences. REE uses *tcmalloc* for memory allocation, which improves the process's performance. It also allows better debugging by introducing the ability to inspect the garbage collector's state and to dump stack traces for all running threads.

This Ruby interpreter is normally paired with *Phusion*'s[7] web server—Passenger. Since they are developed by the same team, some optimizations are more noticeable when these two are being used together.

2.2.1.3 JRuby

JRuby is a Java implementation of Ruby whose first version to support Rails was released in 2006. It has many differences from the MRI and these include [Cora, Loh09]:

Language. JRuby is written in Java, running on top of JVM.

Threading. The thread implementation is based on JVM threads. These are more efficient than the green threads used by MRI, since they are native threads.

Garbage Collector. The garbage collector is inherited from Java, being generational-based. It also inherits its heap and memory management, granting it greater performance since Java's memory management is overall very efficient.

Extensions. JRuby does not support native C extensions, but the most popular ones have already been ported to Java.

Bytecode Interpretation. In this concern, JRuby has the advantage of running on top of the JVM. The Ruby code can be interpreted directly, like MRI's behavior, but it can also be targeted for *just-in-time* or *ahead-of-time* compilation to Java bytecode, which is handled very efficiently by the JVM.

Other Differences. Continuations and forks are not supported in JRuby. There are also a few small differences around its native *endians* and time precision. It does, however, support the Ruby 1.8 specification to full extent and is currently used in many production environments.

2.2.2 Ruby 1.9

This version represents a step forward in the Ruby programming language. It includes many new features and enhancements. Some of these include *Fibers* and *Non-blocking I/O* improvements [cha].

2.2.2.1 YARV

Yet Another RubyVM, also known as YARV, has been adopted as MRI's successor, becoming the official interpreter for version 1.9. For this same reason some people call it KRI or *Koichi's Ruby Interpreter*. It consists on a bytecode interpreter designed specifically for Ruby and it is the only to fully support version's 1.9 specification. Its author purpose, upon its creation, was to reduce execution times of Ruby programs and it differs a bit from other implementations [Sas05, int, FAFH09]:

[7] http://www.phusion.nl/about.html

Language. YARV was developed in C and reuses many parts of its predecessor, like the Ruby script parser, the object management mechanisms and the garbage collector.

Threading. Contrary to MRI, YARV supports native threads. It also efficiently supports *Fibers*, previously called *Continuations*, without suffering from serious performance issues like its predecessor [Sch].

Garbage Collector. As aforementioned, YARV reuses MRI's code for its garbage collector.

Extensions. YARV supports its predecessor's extensions either they are written in Ruby or C. This represents, however, a bottleneck on parallel computing because of existing extensions' synchronization issues.

Bytecode Interpretation. As mentioned before, the main difference between MRI and YARV when it comes to performance is related to the last's generation of intermediate code, which is much faster to process than parsing the program's syntax tree nodes one by one.

Other Differences. There are many differences between YARV and MRI, the reference interpreter. An important one is the usage of *Fibers* which allows the developer to do cooperative scheduling instead of using the preemptive context switch model, commonly used in thread scheduling. On the same subject, *Fibers* are cheaper to create than threads.

YARV initially had a great impact in the Ruby community for its enhancements. It brings many advantages as new features, better performance and improved memory usage. However, it has not been extensively adopted because of some existing incompatibilities with some libraries which have not been upgraded to comply with Ruby's new specification [Can09d].

2.2.3 Promising interpreters in heavy development

There are many *work-in-progress* implementations of interpreters of the Ruby programming language. Some of them are explored in this section.

2.2.3.1 Rubinius

Rubinius has a great deal of focus on performance and its features include support for native POSIX threads, a generational garbage collector, compatibility with MRI/YARV extensions and a more efficient bytecode compiler [EY, Can08b, Can08a]. Unfortunately Rubinius only supports 93% of the Ruby specification, according to the pass rate test from *RubySpec*[8]. However, this interpreter is very promising and its compatibility with the language specification is constantly growing.

[8] http://rubyspec.org/

2.2.3.2 MacRuby

MacRuby is mainly based on YARV. By using Objective-C's engine, it includes a generational garbage collector and supports native POSIX threads. While being a promising Ruby interpreter specifically designed for the Mac OS X operating system, it has yet to achieve an acceptable degree of compatibility with Ruby's specification, which is currently at 85%. Unfortunately, the current version is not able to run a standard Rails 3 application without specific modifications [San10] but a great deal of effort is being put into increasing its compliance with Ruby's specification.

2.3 Rails Web Servers

In its simplest manner, a web server is a never ending *loop* that accepts connections on a listening *socket* and handles them somehow. There are notorious differences on how this *loop* is implemented, besides the classical architectural and philosophical differences behind each web server. These differ in handling multi processing, multi threading, asynchronous events, data copying, context switching, locking contention, memory management, blocking operations, HTTP parsing, the TCP stack implementation and many other architectural differences.

2.3.1 WEBrick

This is Ruby's pioneer web server. It was created in 2000 by Masayoshi Takahashi and Yuuzou Gotou. WEBrick is a full-featured server that supports HTTP, HTTPS and listening concurrently to several ports, among other features. It is purely written in Ruby and has a very modular design, allowing developers to extend its functionalities by supporting external handlers [San04]. WEBrick uses a single process but spawns a new thread for each incoming request. Mainly due to being written in Ruby, its HTTP parser is known for its poor performance [Ali09]. WEBrick's request handling is demonstrated on figure 2.1. Due to its poor performance, users normally used alternate, less conventional setups which were also known for their poor stability [Ali09].

2.3.2 Mongrel

Mongrel was released by Zed A. Shaw in 2006 and soon became the most popular web server used to run Rails applications. It offered a much better performance when compared to WEBrick and it was reasonably suited for production environments. This was mainly due to its improved implementation of the HTTP parser, which was rewritten in C [SN07].

Similarly to WEBrick, Mongrel uses a single process. It has an acceptor thread which handles incoming connections, launching a new thread for each one of them. In production environments, Mongrel is commonly found in clustered configurations where several processes are launched and their usage is dictated by a proxy server [Shaa]. Mongrel's request handling is demonstrated on figure 2.2. Mongrel also optimized the TCP stack by changing Ruby's default socket listening queue from 5 to 1024, besides using optimization flags on socket connections to improve bandwidth usage [Shaa].

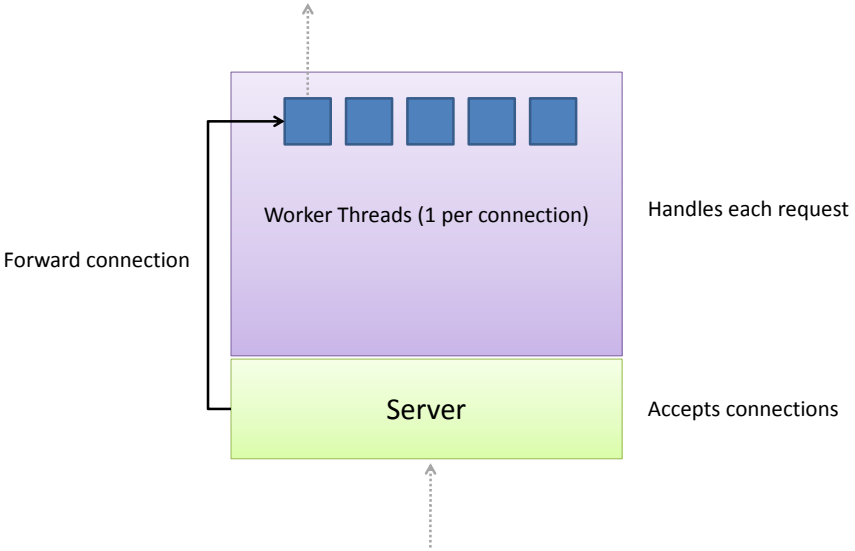

Figure 2.1: WEBrick's Request Handling Process

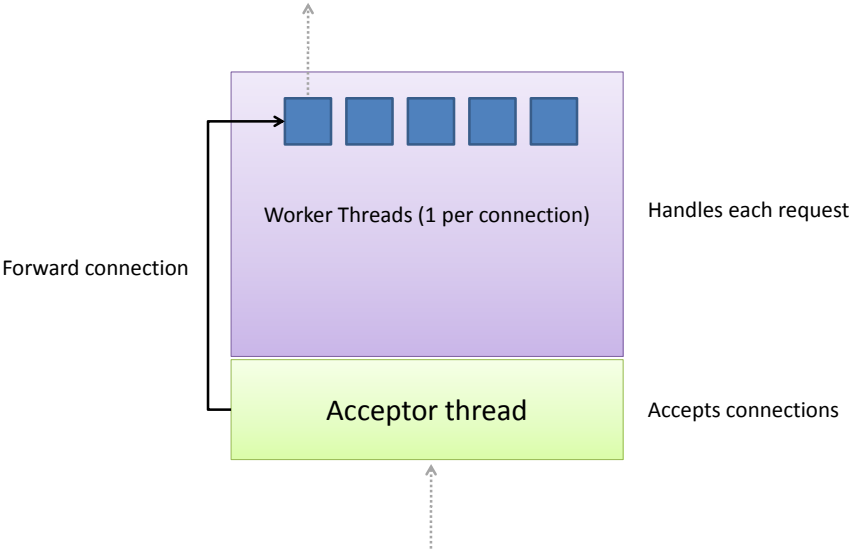

Figure 2.2: Mongrel's Request Handling Process

2.3.3 Thin

Thin was released in 2008 and was the first Ruby web server which did not follow the *one thread per request* convention. It uses Mongrel's HTTP parser and EventMachine as its I/O back-end, allowing it to use a fast asynchronous event loop in a single thread for all incoming requests [Cou]. Thin recently became able to combine threading with its philosophy, by allowing the creation of a background pool of 20 threads [Ali09]. Thin is written in C, C++ and Ruby and is optimized for small requests and fast clients. Its request handling is demonstrated on figure 2.3. This setup yields better performance and scalability than Mongrel, especially when serving small requests like, for example, API calls. This is mainly related to the fact that this web server does not launch a new thread for each request, requiring less memory and no context switches [Ali09].

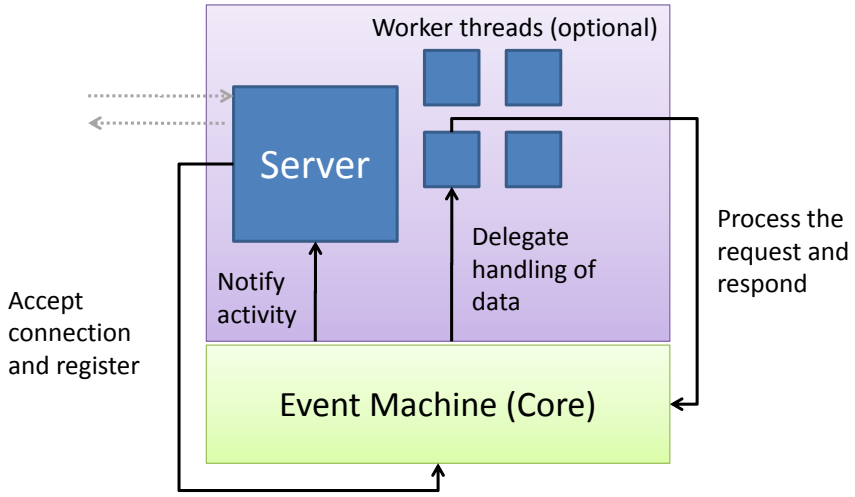

Figure 2.3: Thin's Request Handling Process

2.3.4 Passenger

Passenger was also released in 2008 but it had a big difference from the other alternatives since it was not a self-contained web server. Instead, it makes use of established web servers like Apache or Nginx by using their reliable web stack. It is mostly written in C++ and used as a module or extension to these general purpose web servers, adding the needed functionality to support Ruby and handling certain types of requests [Phua].

When the main web server starts, having Passenger loaded as a module, it launches a Ruby process that will be responsible for all the other processes handling the Ruby application, called "worker processes". Each request is delivered to the firstly created Ruby processes—the master process—which forwards it to one of its workers. These worker processes are single threaded and handle one request at a time [Ali09]. Passenger's request handling is demonstrated on figure 2.4. It is the first real multi-process server for Ruby, although setups with multiple Mongrel or Thin processes behind a reverse proxy were already being used [Phua]. Passenger is a free, open-source product but *Phusion* also provides commercial support.

2.3.4.1 Apache

The Apache HTTP Server is a full-featured and open source web server created by the *Apache Software Foundation*. It consists on a general purpose web server and provides many useful features such as HTTPS, IPV6 and authentication. Apache natively handles many languages such as PHP and Perl [Apa]. It can be extended with modules and this is where Passenger comes in—it will act as *mod_rails* and extend Apache's functionality to be able to handle Ruby on Rails applications [Phua].

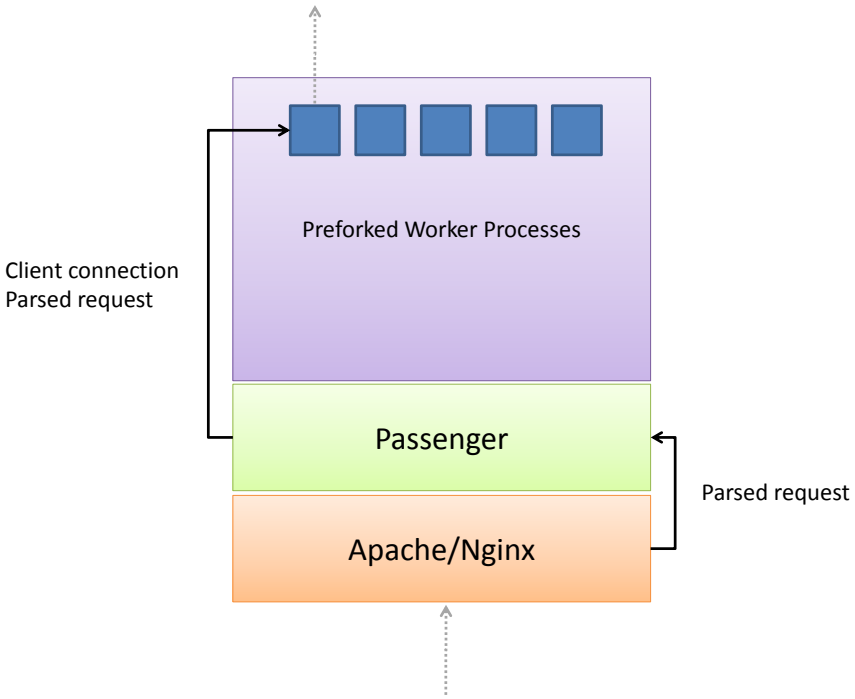

Figure 2.4: Passenger's Request Handling Process

2.3.4.2 Nginx

Nginx is a general purpose lightweight open source web server with a strong focus on performance [Igo]. It was created by Igor Sysoevy and Passenger can extend its functionality by being installed as a module, similarly to the Apache's procedure [Phua].

2.3.5 Unicorn

Unicorn's first stable version was released in 2009. It is a self-contained web server designed to take advantage of Unix-based kernels and is optimized for fast clients with low latency [sig]. It delegates every task that is better supported by the operating system, Nginx or Rack to themselves, respectively. It uses one master process that spawns and reaps a user-defined number of worker processes without any thread usage. One of its main features is that load balancing is done entirely by the OS kernel, avoiding that requests pile up behind a busy worker. Unicorn is written in Ruby, except for its HTML parser which is based on Mongrel's and, consequently, is written in C. When used in a production environment it should be deployed in conjunction with a reverse proxy capable of fully buffering both the requests and responses between itself and a slow client. Unicorn's request handling is demonstrated on figure 2.5.

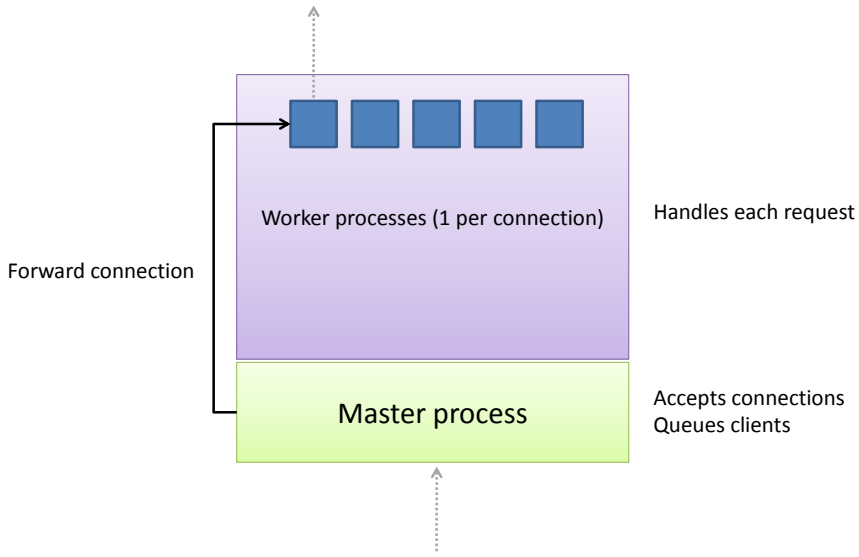

Figure 2.5: Unicorn's Request Handling Process

2.4 Databases

A database is a digitally organized collection of data. It can either be a schema-based or a schema-less database.

2.4.1 MySQL

MySQL is the most popular open source database in the world, having consistently fast performance, high reliability and ease of use [Corc]. It was first released in 1995 and it is the most commonly found database on a Ruby on Rails project. This is the database of choice for all *37signals*'s applications [Gri06]. It is a relational schema-based database that offers useful features like various storage engines, transactions, indexes, load balancing and many others.

MySQL's architecture consists in three main layers. The top one is related with the services that are not unique to MySQL, like connection handling, authentication and security. The middle layer refers to crucial MySQL features like query parsing, analysis, optimization, caching and all the built-in functions. This layer also holds all functionality across storage engines and stored procedures. Finally, the bottom layer consists in the storage engines themselves, responsible for the storage and retrieval of all stored data [SZT+08]. MySQL has four main storage engines—MyISAM, Heap, BDB and InnoDB—with distinct advantages and disadvantages. However, InnoDB is the only storage engine supported in Rails applications. MyISAM and Heap lack transaction support and BDB does not ensure referential integrity, features used by the Ruby on Rails framework. Nevertheless, InnoDB is the default storage engine on MySQL installations.

Ruby has an official library for this database, called *mysql*. There are, however, a few alternatives worth mentioning, namely *mysql2* and *mysqlplus*. These two have some similar and a few other distinct goals:

mysql2 aims at performing the necessary type conversions between MySQL and Ruby types in C and allows asynchronous queries.

mysqlplus aims at supporting asynchronous queries and enabling threaded database access.

The installation of *mysql2* automatically patches ActiveRecord, allowing a smooth transition from the default library. The other mentioned alternative, *mysqlplus*, also replaces the default driver and does not need patching to natively interact with ActiveRecord.

2.4.2 PostgreSQL

PostgreSQL is the most advanced open source database server. It was started by Michael Stonebraker at the University of California in Berkeley and had its first release in 1989. It is a DBMS that contains all features found on other open source or commercial databases and a few more [MS05].

PostgreSQL has some prominent users, like *MySpace*, who strengthen its credibility as a full-featured scalable highly-reliable relational database [Cec09]. Rails' ActiveRecord natively supports this type of database, using Ruby's official library—*ruby-pg*.

2.4.3 MongoDB

MongoDB is a scalable, highly performant, open source, schema-free, document-oriented database written in C++ whose first release was in early 2009. It is a combination of key-value stores, fast, highly scalable and traditional RDBMS systems which provide structured schemas and powerful queries.

This database is document-oriented, providing the simplicity and power of JSON-like data schemas. It supports dynamic queries and indexes. It also provides complex features like replication, auto-sharding and MapRedux [10gb]

This database has been gaining popularity within the Rails community for its simplicity of use, high performance and many features that fit well within the Ruby development philosophy [Nun09]. Mongo is very performance-oriented and some of its features that provide outstanding performance are [10ga]:

- Client driver per language: native socket protocol for client/server interface;
- Use of memory mapped files for data storage;
- Collection-oriented storage (objects from the same collection are stored contiguously);
- Update-in-place;
- Written in C++.

Rails does not natively support MongoDB. For its usage in Rails the developer must replace its default ORM, *ActiveRecord*, with *MongoMapper*. This library provides access to Mongo database operations and natively supports Ruby objects without conversions [Rob09].

2.5 Ruby on Rails

David Hansson began developing a web-based project management tool oriented towards small teams in 2003. Working at *37signals*, he initially started by using PHP but soon became frustrated by many of the language's shortcomings. He gave up on this language and started implementing what today is known as *Basecamp* in pure Ruby. While developing the application, he noticed that a lot of its code could be extracted into a framework for future use with other applications. Hansson decided to release his framework to the public in July 2004 and Ruby on Rails was born [Wil07].

Rails started to become more mature over time and applications like Twitter, YellowPages, Hulu, Scribd and GitHub were built using this framework. Its popularity has grown significantly since the beginning and its adoption by popular platforms helped establishing Rails as a solid framework.

Ruby on Rails has three main principles behind its creation [THB+06, BK07]:

Convention over configuration. In Rails, everything has a default configuration. The only exception is the database connection data. This way, developers only need to specify when they want to use unconventional configurations, offering simplicity while retaining high flexibility.

Don't Repeat Yourself. Also known as DRY, this practice implies that the similar code snippets do not exist in separate locations. Every piece of knowledge is unique, definite and has a relevant representation. This simplifies modifications by the avoidance of having to change the same logic in different parts of the project, allowing applications to keep a high consistency degree.

Model-View-Controller. Rails follows the MVC architecture pattern, keeping the source code well organized by clearly separating the code according to its purpose. The *Model* is responsible for maintaining the state of the application and for specifying the constraints its related data has to obey to. The *Controller* receives the users' input, interacts with the model and finally renders a view page as the result. The *View* can have multiple formats, from JSON to XML, and is essentially what is displayed to the users. This principle's schema is presented in figure 2.6.

Rails' functionality can be altered and extended with *plugins* and *gems*. While being a full stack web framework, Rails does not aim to include every single feature. However, it has been built with a highly extensible infrastructure and it has a considerably large set of *plugins* and compatible *gems* nowadays [Yer09].

2.5.1 Rails 2

Rails 2 was first released in 2007. Its current version is 2.3, released at the beginning of 2009. Throughout these 2 years the framework was improved by many contributors, aside from the core team [Shab]. The framework is essentially divided in 6 modules [BK07, Rai]:

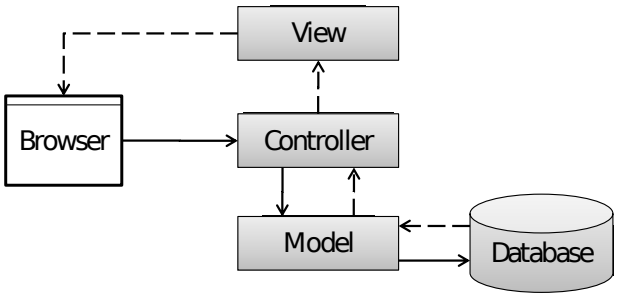

Figure 2.6: Model-View-Controller Architectural Pattern

ActionPack splits the response in two: a request for the controller to handle and a template rendering part for the view to control.

ActiveRecord is responsible for object handling and their database representations. Objects are directly linked to the database, so modifying them will modify the table definition they are associated with.

ActiveResource corresponds to objects that represent the application's RESTful resources as manipulatable Ruby objects.

ActiveSupport is a collection of various utility classes and standard library extensions.

ActionMailer is a framework for designing email-service layers, allowing the application to send emails using mailer models and views.

Having a considerable amount of modules and classes, Rails is structured according to the aforementioned components.

2.5.2 Rails 3

Rails 3 is currently in development and it is on its first release candidate, after four beta versions. Most of Rails' code has been refactored and this release's main goals were concerned with improved component decoupling and performance [Yeh09].

As of component decoupling, a great deal of work has been done and impressive goals have been achieved [Kat09a]. Most of Rails' components became agnostic, having standard interfaces for communicating with each other. The key concept is that a component is agnostic to whom it is interacting with. This allows component swapping, enabling the replacement of one or more of Rails' core components with a different, third-party one. In order to make this happen standard procedures have been developed, providing standard interfaces for each one of Rails' components.

The decoupling process also allowed for improved modularity, permitting Rails' component division. ActionController, for instance, has been split into ActionDispatch, ActionController and AbstractController [Kat09a]. There was a lot of work on explicitly handling each component's internal dependencies. This allows the developer to carefully select which modules he needs in his Rails application without caring about including their dependencies as well. In previous versions

of Rails, developers would generally import the top-level modules, since the alternative was to parse the source code of the framework to find its internal dependencies in order to import all necessary modules. Applications now have the possibility to load only the modules they really need thus becoming faster and lighter.

This improved modularity had its impact on performance. However, the team also made a specific effort into improving common Rails bottlenecks, like partial and collection rendering [Kat09a], among some other optimized sections.

Chapter 3

State of the Art

Each component's state of the art will be presented in this chapter, providing an analysis of related work.

3.1 Operating Systems

This research is focused on server environments since deployed Ruby on Rails applications are commonly found in these setups. When it comes to servers, though, the emphasis on performance becomes much more critical since it is a very important requirement that applications demand from this kind of environment. Some facts suggest that the most commonly found performance bottlenecks in servers are related to the operating system. Its performance bottlenecks usually reside in three key issues [YK94]:

- Process management;
- Virtual memory management;
- High performance I/O.

Each operating system addresses bottlenecks differently and their decisions impact application specific and system-wide performance.

In this research's context, performance is addressed from a Ruby on Rails perspective. Windows' support for this framework and related components is poor and this environment makes the framework slower. Passenger, one of the major Rails web servers used in production, opted not to support the Windows platform. As Marcus Koze explains [Koz08]:

> "We have no plans to port Passenger on Windows. Windows lacks the proper facilities to implement Passenger efficiently. Passenger on Windows will be very, very inefficient, which can give both Ruby on Rails as well as Passenger a bad name."

Many other web-oriented benchmarks show that Linux has significantly better scalability and performance compared to Windows when it comes to this type of servers [Zig06, AG]. The difference is quite noticeable: Ruby on Rails is still more efficient in a virtual machine running Linux than in a normal installation of Windows [Cri09].

Documentation on how to use and deploy Rails applications in the Windows platform is also scarce, often needing unconventional methods for being able to run properly [HH07].

Since Rails is built on top of the Ruby programming language, this framework's lack of performance in Windows is also related to the poor performance of Ruby interpreters in this operating system. The official interpreter for Ruby 1.8 is twice as fast in Linux than its Windows counterpart. As of YARV, the Ruby 1.9 official interpreter, it is 70% faster on Linux when compared to its Windows counterpart [Can09c]. Windows-specific Ruby interpreters do not seem to have noticeable performance improvements [KG08] and lack the necessary stability and compliance with the Ruby specification. *IronRuby*, for instance, is an interpreter developed by Microsoft [Inf09] that is faster than Windows' MRI but still slower than the YARV implementation for this operating system. It also presents stability issues, as timeouts are common in a considerable number of benchmarks [Can09a].

On a side note, MySQL's documentation states that there are a few unpleasant details when using this database under a Windows environment [Corb]. First of all, it is only conceded 4000 ports which after being used need two to four minutes before becoming available again. It also has a limited allowed opened files number—2048—restraining its concurrent capabilities. Finally, MySQL uses a blocking read for each connection and Windows' connection error handling is quite poor when compared to Linux's counterpart. As the aforementioned documentation states, all these differences limit the ability to handle multiple concurrent requests and difficult high-load handling. They can lead to performance issues and limit the system's scalability.

To date, there are no specific tests analyzing Ruby on Rails' performance on BSD when compared to other operating systems. However, many web discussions suggest that it is quite similar to the one achieved on Linux which is understandable since they are both UNIX derivates.

3.2 Ruby

The Ruby on Rails framework, as its name implies, is built on top of the Ruby programming language. Rails' performance is highly affected by Ruby and the interpreter choice can lead to great impact on the framework's scalability.

Ruby has many interpreters according to its version. Ruby 1.8 has MRI as its default interpreter and Ruby 1.9 has YARV. There are, however, a few other implementations that deserve some attention. These present different philosophies and implementation details that provide different advantages and shortcomings, as explored in Chapter 2.2.

The official implementation for Ruby 1.8 is generally known by its poor performance [Gri08]. This motivated the development of efficient interpreters for the language. Ruby Enterprise Edition has better scaling abilities than MRI, generally performs better and uses less memory [Wea09a].

JRuby outperforms MRI by a considerable margin [Dym09]. Rubinius, on the other hand, also consistently outperforms MRI. It also performs better than JRuby in most cases by a much smaller margin [Man].

Some people found significant performance improvements in MRI by applying a series of patches and configurations to its garbage collector. Twitter benefited from these patches, roughly improving its overall performance by 30% [Wea09b].

Ruby 1.9 has only one 100% compliant interpreter as aforementioned. However, as reference benchmarking, its performance is better than JRuby's [Dym09] and is paired with Rubinius'. It performs slightly better in some benchmarks but also performs faintly worse on others [Man, Neu08]. It is a huge improvement from version 1.8. As Dave Thomas puts it [TFH09]:

> "It runs faster, it is more expressive, and it enables even more programming paradigms."

It's increased performance over MRI was noticed by the Ruby community which also lead to discussions on whether it was more efficient than Python's counterpart [Can09b].

Regarding YARV's native profiler, it only supports one workflow: enabling it, running the test code and requesting its report, which is returned in a well formated string [mmotRc].

3.3 Rails Web Servers

The first Rails web server was WEBrick, publicly released in 2000. Mongrel came in 2006 and soon after Thin and Passenger followed.

Using the aforementioned web servers, Muhammed Ali did an extensive overview of some characteristics that heavily contribute to web server performance [Ali09]. His analysis is summarily exposed in the following sections, along with a brief overview of how each web server handles them.

3.3.1 Data Copying

When a server fetches a file from the disk or a back end server it usually copies it to kernel space in order to send it to the client. There are ways of sending the data directly without copying it, called *zero-copy* techniques [STT+09].

WEBrick, Mongrel and Thin neglect this issue. Passenger neglects it as well but, on the other hand, it passes file descriptors to worker processes which avoids sending the data to the clients through the parent web server.

3.3.2 Context Switching

This issue is related with context switching between kernel and user space and all the overhead involved in this procedure. System calls and thread exchange are some of the activities that cause context switches.

WEBrick and Mongrel highly rely on threads, which causes many context switches. In Passenger's case, context switches are related to process context switching, since it uses processes instead of threads. Thin efficiently handles context switches—it does not use threads nor processes, reducing the impact of this issue.

3.3.3 Lock Contention

Multiple threads or processes with shared data imply data locking from time to time. If the data is locked it will put the remaining threads or processes trying to use it on hold.

WEBrick's and Mongrel's threads share almost no data, avoiding noticeable issues related to lock contention. Neither Thin nor Passenger share data between their processes and threads, completely avoiding lock contention.

3.3.4 Memory Management

Web servers allocate and free memory frequently and Ruby's garbage collector is not very efficient for such application behavior. Web servers might counter this issue by allocating a small amount of objects and/or reusing them as much as possible.

WEBrick does not conserve memory neither reuses objects. In Mongrel, though, parser objects are reusable and string usage is kept to a minimum, being replaced by constants. Thin also avoids using strings in detriment of constants but buffers data twice in memory before retrieving it from *Rack*. Passenger has an excellent behavior concerning memory management by using the fork friendly Ruby implementation and by being able to periodically recycle worker processe.

3.3.5 Blocking Operations

I/O operations, as mentioned before, block the server until its completion.

WEBrick solely relies on non-blocking I/O. Contrary, Mongrel performs all I/O in a blocking manner because of its usage of threads. Thin, on the other hand, does most of its I/O operations in a non-blocking manner and Passenger is not affected by this issue since it uses single threaded processes.

3.3.6 HTTP Parsing

The implementation of the HTTP parser greatly affects the web server performance since it is a heavy procedure.

WEBrick's HTTP parser is written in pure Ruby, having overall poor performance. Mongrel, in contrast, uses a fast and secure implementation written in C. While Thin uses the same parser found in Mongrel, Passenger does not parse HTTP as it handles that responsibility to its underlying web server.

3.3.7 TCP Stack

Web servers can tune TCP operations by specifying a set of options allowed by the TCP protocol implementation.

WEBrick and Thin do not try to optimize the TCP stack while Mongrel uses two configuration options that increase the number of allowed waiting connections and optimize bandwidth usage. Finally, these kinds of optimizations on Passenger depend on whether its underlying web server is using TCP stack optimization-oriented configurations or not.

From all the web servers mentioned in this analysis, Passenger presents itself as the easiest to setup in a server environment [Ali09] since it can be installed as a module for other easily installable web servers like Apache or Nginx.

3.3.8 Benchmarks

Some benchmarks were conducted by Muhammed Ali [Ali09], whose results are showed on table 3.1. All tests consist in serving dynamic requests from a single Rails process. All requests are

Table 3.1: Web Server Benchmark Results

TEST	USERS	REQUESTS	WEBRICK	MONGREL	THIN	PASSENGER
1	10	1000	169	438	650	497
	100	1000	209	451	613	509
	1000	1000	250	446	615	324
2	10	1000	115	309	365	305
	100	1000	165	299	350	316
	1000	1000	194	284	346	307
3	10	1000	41	44	43	45
	100	1000	41	42	45	44
4	10	1000	41	37	55	79
	100	1000	41	37	54	79
5	10	1000	6	5	7	10
	100	1000	6	5	2	10

made concurrently and each web server's results are in number of requests handled *per second*. Test 1 is a sample "Hello World!" page. Test 2 fetches a database record and renders it as an ERB template. Test 3 marshals 500 complex objects. Test 4 sends a large 1MB response. Finally, test 5 sends a very large 10MB response.

According to Muhammed's analysis, WEBrick is evidently slow when compared to its counterparts. Mongrel performs much better that its ancestor and both Thin and Passenger improve over it. Thin is clearly the best web server for serving small requests. Passenger, though, is the one to scale better since it performs similarly with 10 or 1000 concurrent requests. It also shows better performance when handling large responses. Its performance could be different if it was using Nginx as its back-end, instead of Apache.

Unicorn was not contemplated on the previously shown tests but it has been gaining reputation and its adoption is increasing. *Twitter*, the popular Rails-based platform mentioned before, recently switched over to Unicorn as its web server, publicly praising some of its features. Among them are its request queue handling, master/slaves architecture, error encapsulation and recovery, CPU usage and a feature called *zero-downtime deployment* that "works beautifully" [BS10].

3.4 Databases

The database is responsible for persisting data. It stores information so it can be retrieved later without needing to use the system's memory. The database usually represents a worrying bottleneck for highly performant applications. It interacts with the disk—the system's main memory—and this generally represents a performance bottleneck [BKM08]. Accessing the hard drive is a costly procedure and web applications need a very efficient database and configuration in order to diminish this bottleneck's impact.

Across most relational databases—from MySQL to PostgreSQL and including proprietary products—MySQL consistently outperforms all others [SMB08].

Despite being the fastest schema-based database in production, a schema-less database like, for instance, MongoDB, often improves over MySQL's performance significantly. This kind of database is more flexible but it also uses more disk space, as row names are stored along with their content to enable this flexibility. MongoDB is not atomic, however, not enabling useful features like transactions [Ice09]. This is not related to the fact of being a schema-less database but with the databases' philosophy. Contrary, CouchDB—another well-known schema-less database created by the *Apache Software Foundation*—supports transactions, although under a performance penalty.

A critical, widely researched approach to improve a system's scalability is database caching. The reduction of disk access generally provides remarkable performance increases [Hof06], allowing improvements greater than 100% [NS06, LCH+01].

Alternative Ruby database adapters can significantly increase performance on database accesses. The *mysql2* library, for instance, can decrease database access times by nearly 400% [Lop].

3.5 Ruby on Rails

Rails is developed by a huge community and coordinated by its core team [Shab]. However, many companies like *Twitter* had the need to squeeze this frameworks' scalability and focused on some critical bottlenecks which had been identified before, like Ruby's garbage collector mentioned in Section 3.2.

There have been some outside performance improvements over Rails 2.3 code, though. From simple Rails' source minor patching [Ice08] to complete architectural makeovers [MGL+07], Ruby on Rails' improvements were widely attempted and some succeeded.

However, the main performance improvements targeted at Rails 2.3 can be found in Rails 3. The decoupling mentioned in Section 2.5.2 significantly improved Rails performance and scalability, allowing faster execution times and lighter memory usage.

There were also specific performance optimizations, namely on *partial* and *collection* handling. Initial performance benchmarks show that partial and collection rendering and overall performance is more than two times faster [Kat09a, Kat09b]. Controller-related code was also refactored to become lighter and more modular. So was *ActiveRecord*, which now uses *Arel*—a relational algebra framework—to generate smarter and more efficient queries before dispatching them to the database adapter.

Regarding Rails' native profiling tools, benchmarking Rails applications under Ruby 1.9 is broken, since it relies on Ruby's *GC::Profiler* "data" method [Han10] which is missing and therefore not available [mmotRc].

Chapter 4

Problem Statement

Ruby on Rails applications generally yield scalability issues. A significant amount of applications developed using this framework exhibit this problem by publicly starting to suffer from performance and scalability problems as their popularity increases. The Ruby on Rails community is not generally focused in building highly performant systems. The lack of centralized information, functional and intuitive profiling tools and global awareness of this subjects' importance all contribute to this problem.

As observable on Chapter 3, some information regarding benchmarks and configurations of the various components involved in a Rails application exists but it is, however, very scattered. Most tests and configuration evaluations are not general, focusing on specific components, setups or purposes. It is crucial to create a solid set of general conventions and guidelines that cover all components, approaching benchmarking and tweaking from a unified perspective—optimizing a Ruby on Rails application.

On a related matter, the native profiling tools in Ruby on Rails have never worked properly when used with Ruby 1.9. While difficulting the act of profiling an application, this also prevents some users from switching to this version and leveraging from its increased performance. At the same time, the profiling output formats supported are all text based [Han10]. Profiling is an essential aspect of increasing an application's performance, so it is very important to fix Ruby and Rails' profiling tools, provide a seamless integration between them and the most recent versions of Ruby and add support for alternate, more intuitive profiling output formats.

As mentioned in Chapter 1, Twitter—and a few other renowned platforms—had many scalability issues which were not discrete, some becoming quite famous. However, the global awareness of the Ruby on Rails community regarding the importance of scalability is still remarkably low. Despite having the Rails 3 API available shortly after its development began back in early 2009 and Ruby 1.9 available since 2008, most developers have not added support for these versions to their plugins and gems. Famous Rails applications, like Redmine, also lack support for Ruby 1.9 and have not started upgrading to Rails 3. Knowing the performance benefits of using the latest

versions of both components, it is important to increase the community's *momentum* by updating renowned plugins and applications, possibly igniting an update-focused philosophy.

To create a general guideline with centralized information, improve the current profiling tools and increase the community's sensibility on this subject, it is crucial to address specific characteristics, configurations and issues of all components involved in a Rails application using the aforementioned perspective.

In order to build highly performant systems, developers need to know which Operating System is best suited for Rails applications. It is also important to determine the benefits of upgrading to the most recent Ruby version, improve one of its main Rails-related bottlenecks—the garbage collector—and greatly enhance its profiling capabilities. Developers should also be aware of the raw performance, high-load stability and memory usage of the recent versions of renowned Rails Web Servers. The official database adapter for MySQL is written in Ruby and generally inefficient, posing as an excellent target for improvements concerning Databases. It is also crucial to enhance Rails' profiling tools and to make them seamlessly integrate with Ruby's. Finally, to motivate the early adoption of this framework's latest version—Rails 3—the porting effort should be decreased and, on the other hand, the awareness of its advantages should be increased.

The following sections state and justify the problems to be addressed in each component.

4.1 Operating Systems

Focusing on finding specific Operating System bottlenecks and solving or improving them is beyond the scope of this project. It is also unlikely to be worthwhile given the time span of the project and its wide area coverage. Furthermore, most operating systems have significant communities discussing, implementing and improving solutions to given bottlenecks.

However, there is the need to determine which OS is best suited for Rails applications. The lack of general benchmarking tools limits one's ability to fairly compare the performance of operating systems. These should be created, taking into account common tasks performed by them.

Ascertaining which OS best handles web server load with the default configurations is also important to build a starting point for deploying and optimizing Rails applications.

Finally, as mentioned in Section 2.2, Ruby is mainly developed in Linux but should easily run on many other operating systems. However, as seen in Section 3.1, its performance can greatly differ depending on what OS is running it. The analysis of the performance differences of the Ruby interpreters across operating systems should continue, aside from those already determined in other recent analysis.

4.2 Ruby

As mentioned in Section 3.2, YARV has noticeably better performance than its ancestor. Most Rails applications were developed with version 1.8 and would highly benefit from the new version's improvements if upgraded. It is crucial to determine the benefits of this upgrade to justify

the effort needed to accomplish it.

As explained in Section 3.2, most of the Ruby-related performance bottlenecks found in Rails applications are related to its garbage collector. Besides using a garbage collection algorithm—*mark-and-sweep*—whose performance is below other algorithm's, it is not configurable, lacking the ability to adapt itself to the application it is running. As seen in Section 2.2, there are many benefits in having a configurable GC, so it becomes important to give YARV this flexibility.

Finally, the integration between Rails' profiling and benchmarking tools and Ruby is outdated or, in some cases, nonexistent. It is important to update the existing tools, increase the amount of recorded data and include new output formats, therefore improving YARV's data retrieval abilities and its integration with Rails' profiling and benchmarking facilities.

4.3 Rails Web Servers

There are many analysis concerning web server performance but unfortunately most of them are either outdated, not applicable to nowadays' setups, do not cover a wide range of web servers or lack relevant data. For instance, the high quality analysis mentioned in 3.3 does not include Unicorn, nor does it test Mongrel, Passenger and Thin behind other web servers. Furthermore, one of the critically significant but frequently missing aspect in benchmarks is memory usage.

It is also important to analyze the high-load response of each web server by stressing it to its limits and evaluating its behavior from a stability perspective.

An up-to-date analysis covering raw performance, high-load stability and memory usage is crucial for increasing developers' awareness of the benefits and shortcomings of each web server setup. Since most web servers are natively flexible and configurable, its essential to explore and analyze their configuration options as well.

4.4 Databases

Similarly to the OS component, focusing on finding specific database bottlenecks and solving or improving them is beyond the scope of this project. It is also unlikely to be worthwhile given the same reasons stated in Section 4.1.

However, Ruby plays an important role concerning databases. The current implementation of the *mysql2* library converts the data between MySQL types and Ruby objects immediately after fetching each row from the database. This behavior is not optimal for situations where there are more fields being fetched than those being used. Changing the program's flow in order to trigger type casting only when the variable is actually accessed—lazy type casting—can have a significant impact on the aforementioned conditions, becoming an important issue to address.

This library is likely to replace Ruby's default in the future. It is being actively developed and provides significant ameliorations from the current library, *mysql*. It is starting to gain a solid reputation within the Ruby community, posing as an excellent target for improvements concerning this component.

4.5 Ruby on Rails

The current native profiling tools for Ruby on Rails are deprecated. They do not function properly on up-to-date environments and they highly depend on Ruby's profiling abilities which are very limited. Improving Ruby's profiling tools is not enough as they need to seamlessly integrate with Rails'. For this to happen, Rails must also be targeted for improvement by refactoring and improving the existing non-functional profiling tools.

Rails 3's performance is significantly better when comparing to its predecessor, as mentioned in Section 3.5. However, the adoption of Rails 3 is likely to be difficult and lengthy since applications need to be refactored and most plugins—one of the main advantages of this framework as explained in Section 2.5—need to be rewritten. To motivate its early adoption and increase this version's *momentum*, some of the most renowned plugins need to be ported to the new version. Getting one of the most famous applications—Redmine—to work properly on Rails 3 would also increase the awareness of the new versions' benefits. Finally, some performance bottlenecks commonly found in Rails applications should also be addressed and simple solutions presented.

4.6 Summary

Rails applications generally yield scalability problems. Developers are not aware of the performance benefits and drawbacks of the choices they have for each component. They are also unable to profile their applications using the official tools, since they are non-functional. Some system components are not fully optimized and can be significantly improved. All these issues span into the multiple system components involved, which must be addressed from a performance standpoint and from an unified perspective—Ruby on Rails.

Chapter 5

Problem Approach and Results

Envisioning a system-wide performance optimization of a Ruby on Rails application requires targeting all the components involved from the previously mentioned centric perspective—Ruby on Rails. Conducting benchmarks, tweaking configurations, developing improved solutions and evaluating the results must be accomplished with this philosophy in mind.

It is also necessary to perform these activities using a fair base of comparison. The most important rules used when benchmarking were:

Same tests: Use the same tests when testing component alternatives and, when impossible, keep their disparities to a minimum.

Same hardware: Use the same hardware when testing a given component.

Similar configurations: Use equal or, at the very least, similar configurations for all the components involved. Exceptions can be made when the purpose of the test is to benchmark the behavior with the default configurations.

Two distinct machines were used during this research. Due to hardware malfunction, the initial computer had to be replaced with a different one. "Machine 1" one was used on all the OS-related work, while "Machine 2" was used for all the activities related to the remaining components. Their specifications are listed on table 5.1.

Table 5.1: Hardware Specifications of the Machines in Use

	MACHINE 1	MACHINE 2
CPU	Intel Core 2 Quad Q9300 @ 2.50GHz (FSB @ 1333MHz)	Intel Core 2 Duo E8400 @ 3.0GHz (FSB @ 1333MHz)
RAM	2x2GB DDR2 (800 MHz, Dual Channel)	4x2GB DDR2 (800 MHz, Dual Channel)
HARD DRIVE	Seagate ST3500620AS 500GB SATA, 16MB Cache	Seagate ST3750630AS 750GB SATA, 16MB Cache

When benchmarking, the same software versions were used across all systems and environments. Table 5.2 shows the software versions that were used. All packages were compiled from source using GCC with *"-O2 -march=nocona -pipe"* as their compilation flags.

Table 5.2: Software Versions in Use

SOFTWARE/PACKAGE	VERSION
MRI	1.8.7 (patchlevel 249)
YARV	1.9.1 (patchlevel 378)
Rails	2.3.5
FreeBSD	8
Linux Kernel	2.6.26
hdparm	9.15
gzip	1.3.12
lame	3.98.2
vorbis-tools	1.2.0
Apache	2.2.14
Nginx	0.7.64
Cherokee	0.99.42
Thin	1.2.7
Unicorn	0.97.0
Passenger	2.2.11
autobench	2.1.2
httperf	0.9.0
GCC	4.3.4

To achieve the objectives established in Chapter 4, some activities were performed.

First of all, to create the centralized set of conventions and guidelines for improving the performance of Ruby on Rails applications, there is the need to analyze and benchmark all worthy alternatives for each component.

In order to easen the process of profiling Rails applications, the native profiling tools must be fixed and upgraded. Rails and Ruby must be improved and tweaked to accommodate all changes, providing a seamless integration between each other's tools.

Finally, to improve the global awareness of this issue's importance there is the need to gather the benefits associated with it, provide a smoother transition by updating renowed Rails plugins, increase this subjects' notoriety by updating one of the most renowned Ruby on Rails projects—Redmine—and by publishing a series of articles about performance optimization on the *Rails Magazine* and, ultimately, initiate a performance-oriented continuous integration for Rails by developing a benchmarking suite aimed at the framework itself.

In order to accomplish the aforementioned tasks, every component was specifically addressed. The work done is presented and explained in the following sections.

5.1 Operating Systems

Using Linux and BSD, the focus on this system component was to create a generic benchmarking script, determine the operating system in which common web servers perform better and determine the OS in which the official Ruby interpreters have the best performance.

As exposed in Chapter 3.1, Windows is not a suitable OS for production environments of Ruby on Rails applications because of its poor and inefficient support for applications that use this framework, being excluded from further research. On the other hand, Mac OS X Server requires specific hardware so any comparison would not be rigorous, also being excluded from further research. However, its performance is expected to be similar to BSD systems since, as mentioned in Section 2.1, its kernel is based on this OS.

The work on the remaining components will be conducted using the OS and distribution which yields the best results in this phase.

Ascertaining which OS is best suited for Rails applications was the general goal of the work presented in this section.

5.1.1 Development

Concerning development, a generic benchmarking script was created. This script was based on a few commonly found tools on UNIX setups and consists on 5 micro tests and 1 macro test, respectively:

1. Use hdparm[1] to time cached reads on the disk;
2. Compress a 2.5GB file to ZIP format using gzip[2];
3. Uncompress the previously created archive;
4. Convert a 214MB WAV file to MP3 using the lame MP3 encoder[3];
5. Convert the same 214MB WAV file to OGG using vorbis-tools[4];
6. Parallelly run all the aforementioned benchmarks while extracting, compiling, installing and removing PHP[5] 5.2.12.

The first test aims at asserting hard drive access speed, which are dependent on the filesystem in use and the OS's I/O management. The second and third tests are more complex but have a similar purpose. Both read a file with considerable size from the disk, convert it and write the result. However, the main bottleneck happens when writing the result file since writing is slower process than reading and the ZIP algorithm is lightweight and fast. The fourth and fifth tests are more CPU-intensive, since they convert a file between different audio formats. The tools in use—*lame* and *vorbis-tools*—stress the OS even further by using multiple processes and threads, inducing various context switches. Finally, the last test aims at determining the OS's ability to manage a high workload since multiple heavy tasks are being carried simultaneously, involving concurrent I/O, multiple context switches, scheduling and a few other core tasks.

[1] http://sourceforge.net/projects/hdparm/
[2] http://www.gzip.org/
[3] http://lame.sourceforge.net/
[4] http://vorbis.com/
[5] http://php.net/

The goal is to measure the amount of time—user plus system time, not accounting waiting periods—needed to accomplish each task. The number of voluntary context switches and the average CPU usage are also recorded, though as secondary data. GNU time[6] is used to make all measurements except in the first test, since *hdparm* itself measures the amount of cached data which is read in 2 seconds, yielding results in MB *per second*. All tests are ran a configurable amount of times—the default being 3—to cancel circumstantial issues. An exception is made on the last test which is very lengthy, therefore canceling circumstantial issues by itself. It runs only once.

5.1.2 Benchmarking

The benchmarking phase had the clear goal of determining which is the best OS to invest in the remaining work. It was also very important to gather data about each OS/distribution's behavior so that it would be inserted in the aforementioned guidelines and conventions.

5.1.2.1 Generic Benchmarking of Linux Distributions

First of all, it was important to choose one of the Linux distributions mentioned in Section 2.1 to be stacked against FreeBSD, the most popular BSD distribution. A benchmark using the aforementioned generic script was performed on Ubuntu Server, Debian, CentOS and Gentoo. All distributions were running their default configurations for all packages. The results are presented in table 5.3 and show the amount of time each test took on each distribution. The amount of voluntary context switches and average CPU usage are also included as secondary information. Gentoo's performance was slightly superior in the first test, yielding results which range from 3% to 7% better than the other distributions. The second test had similar results, with Debian's performance being very close to Gentoo's. CentOS showed serious issues in this test, being approximately 502% slower than Gentoo. Regarding the third test, Gentoo showed superior performance again, with Debian having the second best results. CentOS yields very poor results again, being approximately 1903% slower than Gentoo. Quite unexpectedly, CentOS had the best performance in the fourth test by a significant margin, with Debian's results being the second best once again. Ubuntu yielded the best performance in the fifth test, with Debian and Gentoo having very close results. CentOS shows the worst results by a considerable 13% margin. Finally, Debian yielded the best results in the final test by a substantial margin. CentOS's results, once more, show a considerable performance deficiency when compared to the other distributions' results.

According to these results, Gentoo is the best distribution in CPU usage and I/O operations on a single instance, present in tests 1, 2 and 3. When it comes to almost pure CPU usage, present in tests 4 and 5, CentOS and Ubuntu yielded the best results. Last but not least, Debian showed an impressive behavior handling multiple concurrent tasks, present in the last test.

CentOS's behavior was unstable and inconsistent. Given these results, it was discarded from future work.

[6] http://www.gnu.org/software/time/

Table 5.3: Generic Benchmark of Linux Distributions Result

TEST	UNITS	GENTOO	UBUNTU	DEBAIN	CENTOS
hdparm	MB/s	**2350**	2293	2191	2220
gzip	user+sys/s	**137.75**	147.53	137.89	691.58
	voluntary context switches/n	5878	8111	417	2530
	average cpu usage/%	80%	81%	94%	50%
gunzip	user+sys/s	**22.21**	27.69	23.84	422.72
	voluntary context switches/n	7498	7852	218	4905
	average cpu usage/%	41%	47%	73%	34%
lame	user+sys/s	101.69	101.67	92.18	**83.98**
	voluntary context switches/n	8	7	2	36
	average cpu usage/%	100%	99%	99%	98%
oggenc	user+sys/s	39.09	**37.75**	38.58	42.59
	voluntary context switches/n	7	12	35	178
	average cpu usage/%	99%	99%	100%	99%
workload	user+sys/s	379	318	**202**	1622
	voluntary context switches/n	102525	27035	6491	20537
	average cpu usage/%	151%	129%	118%	58%

5.1.2.2 Web Server Benchmarking on Linux distributions

Since three possible Linux distributions still remained, a different benchmark was conducted. This time, its focus was oriented towards web server performance.

This test used a simple static HTML page served by either Apache or Nginx. Using Ubuntu Server, Debian and Gentoo, many requests/concurrency combinations were used, namely:

- 10000 requests, 1000 concurrent;
- 100000 requests, 1000 concurrent;
- 100000 requests, 10000 concurrent.

Apache's ab[7] was used to perform the tests. Since the goal was to measure raw web server performance on each OS all requests were made locally, providing zero network overhead. If more than 1% of the requests took more than 30 seconds to be replied to the test was considered a failure, as it already surpasses an acceptable response time in real world applications [Nie93]. The web server configurations were not the default ones on this test. Since some distributions loaded more modules than others and this could have a significant impact on the web server's performance, all unnecessary modules and options were removed from the configurations on this benchmark.

Regarding the Apache benchmark, table 5.4 shows that Gentoo had the best performance in the first Apache test, with Debian achieving similar results. Ubuntu, however, did not cope with the others' behavior, needing a considerable amount of extra time to accomplish the same test. As seen on table 5.5, Gentoo showed the best performance in the second Apache test. Debian had a

[7] http://httpd.apache.org/docs/2.0/programs/ab.html

Table 5.4: OS Benchmark Using Apache (10000/1000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	**1.339**	9.755	1.449
Requests *per second* (n)	**7468.32**	1025.13	6901.61
Time per request (ms)	**133.899**	975.490	144.894
Transfer rate (KB/s)	2053.30	453.50	**2966.13**
Average connection time (ms)	29	98	**25**
Minumum connection time (ms)	15	**4**	7
Maximum connection time (ms)	**668**	9755	1444

Table 5.5: OS Benchmark Using Apache (100000/1000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	**14.835**	FAIL	57.382
Requests *per second* (n)	**6740.75**	FAIL	1742.70
Time per request (ms)	**148.351**	FAIL	573.822
Transfer rate (KB/s)	**1849.76**	FAIL	748.84
Average connection time (ms)	**70**	FAIL	330
Minimum connection time (ms)	15	FAIL	**0**
Maximum connection time (ms)	**12050**	FAIL	48966

Table 5.6: OS Benchmark Using Apache (100000/10000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	24.083	FAIL	FAIL
Requests *per second* (n)	4152.25	FAIL	FAIL
Time per request (ms)	2408.331	FAIL	FAIL
Transfer rate (KB/s)	1139.70	FAIL	FAIL
Average connection time (ms)	148	FAIL	FAIL
Minimum connection time (ms)	32	FAIL	FAIL
Maximum connection time (ms)	23422	FAIL	FAIL

considerably worse performance and Ubuntu failed this test as many requests took more than the aforementioned 30 seconds to be replied to. Finally, table 5.6 shows the results of the last Apache benchmark, where all distributions failed to successfully complete the benchmark except Gentoo, which needed a high average amount of time to complete each request but was still able to reply to all requests within the established time limit.

Concerning the Nginx benchmark, table 5.7 shows that it was Debian to achieve the best result on the first benchmark, yielding an impressive performance. The results of the second test are shown on table 5.8 and Debian is the best performing distribution once again, with Ubuntu starting to show signs of not being able to cope with the demand. The third test's results showed some unexpected results since, similarly to the Apache benchmark, neither Debian nor Ubuntu were able to reply to at least 99% of the requests withing 30 seconds, leaving the best result to Gentoo which was the only distribution to successfully complete the final test. The final test results can be found in table 5.9.

Table 5.7: OS Benchmark Using Nginx (10000/1000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	0.833	0.791	**0.540**
Requests *per second* (n)	12001.11	**12647.47**	10000.00
Time per request (ms)	83.326	79.067	**54.013**
Transfer rate (KB/s)	3065.96	4439.88	**6576.40**
Average connection time (ms)	55	15	**11**
Minimum connection time (ms)	40	8	**5**
Maximum connection time (ms)	**75**	256	225

Table 5.8: OS Benchmark Using Nginx (100000/1000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	7.883	14.835	**6.312**
Requests *per second* (n)	12685.55	6740.75	**15842.34**
Time per request (ms)	78.830	148.351	**63.122**
Transfer rate (KB/s)	3168.52	1849.76	**5608.19**
Average connection time (ms)	78	70	**45**
Minimum connection time (ms)	48	15	**7**
Maximum connection time (ms)	**3090**	12050	3674

Table 5.9: OS Benchmark Using Nginx (100000/10000)

	GENTOO	UBUNTU	DEBIAN
Total time (s)	11.555	FAIL	FAIL
Requests *per second* (n)	8654.34	FAIL	FAIL
Time per request (ms)	1155.489	FAIL	FAIL
Transfer rate (KB/s)	2212.22	FAIL	FAIL
Average connection time (ms)	1083	FAIL	FAIL
Minimum connection time (ms)	272	FAIL	FAIL
Maximum connection time (ms)	4473	FAIL	FAIL

Gentoo showed an excellent behavior when scaling. The difference in the average time needed to complete each request on tests 1 and 2 of both web servers is remarkably small. It was also the only distribution to be able to cope with 100000 requests with 10000 of them concurrent, either on Apache and Nginx.

All these distributions rely on the same underlying *kernel*—the Linux *kernel*. Their behavior and performance are similar and the most considerable differences are likely to be circumstantial. However, Gentoo has shown slightly better results at scaling on the web servers' benchmark and, for this reason, it will be the distribution to be compared with FreeBSD.

5.1.2.3 Ruby Benchmark on Gentoo Linux and FreeBSD

Given this research's scope, it is important to determine which of the aforementioned operating systems—Gentoo Linux or FreeBSD—provide the best environment for a Ruby on Rails applica-

tion. This kind of application, as the framework's name implies, is written in Ruby. Therefore, Ruby is a core component from Rails' perspective. The official Ruby interpreters are likely to yield different performance results on different operating systems since they are mainly developed in Linux and then ported to other OS. If we take into account the already known differences stated on Section 3.1, benchmarking this core component is likely to yield different results and to enable a trustful assertion about whether the OS in which it is developed—Linux—is or is not the best for a Ruby on Rails application.

For this analysis, Antonio Cangiano's Ruby benchmarking suite[8] was used. It currently contains 62 micro benchmarks which test specific Ruby features, 8 macro benchmarks which test multiple Ruby features in a single test and 3 RDoc[9]-related benchmarks. This high test variety provides a wide coverage of many Ruby features, solidly asserting about the interpreter's overall performance. Each benchmark had a 300 second timeout and ran 5 times, after which the best run was picked. All tests used both official Ruby interpreters—MRI (Ruby 1.8) and YARV (Ruby 1.9)—in both operating systems.

For readability purposes, each benchmark description indicates if it is a micro, macro or RDoc benchmark. It also states the test ran by using a simplified name which illustrates its purpose. The used format is as follows:

```
[macro|micro|rdoc]/simplified_name
```

Additionally, the input size is also mentioned and its meaning depends on the test itself. For instance, when calculating π it indicates the number of decimal places to determine and when calculating prime numbers it indicates the maximum number to calculate. In order to express the performance differences between the alternatives, a ratio is shown. It expresses the extra time needed to complete the test by the worse performing OS when comparing to its counterpart's result. Finally, a sum which contempts the time of all tests which were successful on both operating system's is shown. All exhibited time units are in seconds.

The following tables will present the amount of time each test needed to complete on each operating system and interpreter, in seconds.

Table 5.10: MRI Benchmark on Gentoo and FreeBSD

Benchmark	Input Size	1.8.7 (Gentoo)	1.8.7 (FreeBSD)	Ratio
macro/cal	500	**1.990**	2.608	31.04%
macro/dirp	10000	**0.386**	0.465	20.53%
macro/gzip	100	**6.141**	6.528	6.30%
macro/hilbert_matrix	10	**0.036**	0.048	33.21%
macro/hilbert_matrix	20	**0.335**	0.431	28.83%
macro/hilbert_matrix	30	**1.367**	1.667	22.01%

Table 5.10 — continued on the next page

[8] http://github.com/acangiano/ruby-benchmark-suite/
[9] http://rdoc.sourceforge.net/

Table 5.10 — continued from previous page

Benchmark	Input Size	1.8.7 (Gentoo)	1.8.7 (FreeBSD)	Ratio
macro/hilbert_matrix	40	**3.866**	4.532	17.23%
macro/hilbert_matrix	50	**8.868**	9.309	4.97%
macro/hilbert_matrix	60	**18.399**	18.489	0.49%
macro/list	1000	**0.053**	0.066	24.16%
macro/list	10000	7.154	**6.291**	13.73%
macro/mpart	300	**0.039**	0.190	383.32%
macro/norvig_spelling	50	**8.562**	11.937	39.43%
macro/observ	100000	**0.612**	0.813	32.78%
macro/parse_log	100	**1.151**	1.200	4.27%
macro/pi	1000	**0.026**	0.030	16.45%
macro/pi	10000	2.127	**2.108**	0.92%
macro/rcs	100	**0.753**	0.788	4.65%
macro/sudoku	1	**10.153**	15.881	56.42%
micro/app_factorial	5000	StackError	StackError	
micro/app_fib	30	**1.587**	2.473	55.83%
micro/app_fib	35	**17.854**	27.183	52.25%
micro/app_mandelbrot	1	**1.899**	2.443	28.65%
micro/app_pentomino	1	SignalException	SignalException	
micro/app_tak	7	**1.173**	1.759	49.92%
micro/app_tak	8	**3.405**	5.003	46.93%
micro/app_tak	9	**8.910**	13.285	49.09%
micro/app_tarai	3	**3.873**	6.021	55.47%
micro/app_tarai	4	**4.670**	7.260	55.47%
micro/app_tarai	5	**5.654**	8.826	56.10%
micro/binary_trees	1	**54.757**	74.456	35.98%
micro/count_multithreaded	1	**0.004**	0.004	2.75%
micro/count_multithreaded	2	0.009	**0.009**	0.30%
micro/count_multithreaded	4	**0.017**	0.017	0.65%
micro/count_multithreaded	8	**0.034**	0.034	0.23%
micro/count_multithreaded	16	**0.067**	0.068	1.96%
micro/count_shared_thread	1	**0.044**	0.050	13.92%
micro/count_shared_thread	2	**0.044**	0.050	13.49%
micro/count_shared_thread	4	**0.044**	0.050	13.31%
micro/count_shared_thread	8	**0.044**	0.050	12.37%
micro/count_shared_thread	16	**0.045**	0.050	12.62%
micro/eval	1000000	**1.843**	2.611	41.64%
micro/fannkuch	6	**0.005**	0.007	32.24%
micro/fannkuch	8	**0.398**	0.466	17.25%
micro/fannkuch	10	**44.995**	49.869	10.83%
micro/fasta	1000000	**37.028**	69.053	86.49%
micro/fiber_ring	10	LoadError	LoadError	
micro/fiber_ring	100	LoadError	LoadError	

Table 5.10 — continued on the next page

Table 5.10 — continued from previous page

Benchmark	Input Size	1.8.7 (Gentoo)	1.8.7 (FreeBSD)	Ratio
micro/fiber_ring	1000	LoadError	LoadError	
micro/fractal	5	**4.520**	4.900	8.39%
micro/gc_array	1	**44.990**	50.502	12.25%
micro/gc_mb	500000	**0.687**	0.883	28.58%
micro/gc_mb	1000000	**1.513**	1.987	31.31%
micro/gc_mb	3000000	**3.535**	5.761	62.96%
micro/gc_string	1	**7.874**	14.727	87.03%
micro/knucleotide	1	**1.396**	1.857	33.03%
micro/lucas_lehmer	9689	4.019	**3.975**	1.10%
micro/lucas_lehmer	9941	4.342	**4.282**	1.42%
micro/lucas_lehmer	11213	6.174	**6.068**	1.76%
micro/lucas_lehmer	19937	33.143	**32.628**	1.58%
micro/mandelbrot	1	**55.460**	62.065	11.91%
micro/mbari_bogus1	1	StackError	StackError	
micro/mergesort	1	**1.914**	2.396	25.22%
micro/mergesort_hongli	3000	**4.423**	5.638	27.47%
micro/meteor_contest	1	**25.352**	35.424	39.73%
micro/monte_carlo_pi	10000000	**12.178**	15.732	29.18%
micro/nbody	100000	**7.046**	7.921	12.42%
micro/nsieve	9	**15.221**	19.085	25.39%
micro/nsieve_bits	8	**19.753**	30.533	54.57%
micro/open_many_files	50000	**0.211**	0.356	68.80%
micro/partial_sums	2500000	**18.003**	19.447	8.02%
micro/primes	3000	**5.057**	7.247	43.31%
micro/primes	30000	SignalException	SignalException	
micro/primes	300000	SignalException	SignalException	
micro/primes	3000000	SignalException	SignalException	
micro/quicksort	1	**7.116**	13.709	92.65%
micro/read_large	100	**6.337**	11.705	84.69%
micro/regex_dna	20	**2.644**	3.843	45.36%
micro/reverse_compliment	1	**3.702**	5.343	44.32%
micro/simple_connect	1	0.134	SocketError	
micro/simple_connect	100	0.143	SocketError	
micro/simple_connect	500	0.175	SocketError	
micro/simple_server	1	0.134	SocketError	
micro/simple_server	100	0.136	SocketError	
micro/simple_server	100000	1.427	SocketError	
micro/so_ackermann	7	0.494	StackError	
micro/so_ackermann	9	StackError	StackError	
micro/so_array	9000	**6.867**	11.425	66.37%
micro/so_count_words	100	**2.408**	2.455	1.93%
micro/so_exception	500000	**8.009**	10.815	35.03%

Table 5.10 — continued on the next page

Table 5.10 — continued from previous page

Benchmark	Input Size	1.8.7 (Gentoo)	1.8.7 (FreeBSD)	Ratio
micro/so_lists	1000	**8.649**	11.233	29.89%
micro/so_lists_small	1000	**1.741**	2.264	30.03%
micro/so_matrix	60	**1.924**	2.821	46.66%
micro/so_object	500000	**1.424**	2.054	44.30%
micro/so_object	1000000	**2.825**	4.104	45.31%
micro/so_object	1500000	**4.265**	6.165	44.56%
micro/so_sieve	4000	**54.784**	58.611	6.99%
micro/socket_transfer_1mb	10000	0.353	SocketError	
micro/socket_transfer_1mb	1000000	0.356	SocketError	
micro/spectral_norm	100	**0.932**	1.433	53.73%
micro/string_concat	10000000	**5.655**	12.219	116.07%
micro/sum_file	100	**9.920**	15.563	56.89%
micro/word_anagrams	1	**7.750**	9.853	27.14%
micro/write_large	100	**0.157**	0.489	212.46%
rdoc/against_itself_darkfish	1	**13.118**	15.236	16.14%
rdoc/against_itself_ri	1	**12.854**	16.533	28.62%
rdoc/core_darkfish	1	SystemExit	SystemExit	
Sum		**700.311**	**905.764**	**29.34%**

As seen on table 5.10, MRI has a consistently better overall performance in Linux. The whole benchmark took 29.34% more time to complete under FreeBSD. Some prominent differences include the “mpar” test which splits a file into multiple parts and the tests entitled “write_large”, “read_large” and “open_many_files”, indicating that I/O in MRI is much faster under Linux than under BSD. Other notable differences include string concatenation, its garbage collection and the “quicksort” test.

Table 5.11: YARV Benchmark on Gentoo and FreeBSD

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1 (FreeBSD)	Ratio
macro/cal	500	0.289	**0.287**	0.48%
macro/dirp	10000	0.393	**0.384**	2.45%
macro/gzip	100	**5.979**	6.441	7.71%
macro/hilbert_matrix	10	**0.002**	0.002	4.98%
macro/hilbert_matrix	20	**0.031**	0.034	9.32%
macro/hilbert_matrix	30	**0.154**	0.166	7.85%
macro/hilbert_matrix	40	**0.477**	0.514	7.71%
macro/hilbert_matrix	50	**1.256**	1.347	7.22%
macro/hilbert_matrix	60	**3.024**	3.206	6.01%
macro/list	1000	**0.026**	0.035	31.92%
macro/list	10000	**2.758**	3.468	25.76%

Table 5.11 — continued on the next page

Table 5.11 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1 (FreeBSD)	Ratio
macro/mpart	300	**0.034**	0.189	461.63%
macro/norvig_spelling	50	ArgumentError	ArgumentError	
macro/observ	100000	**0.360**	0.378	4.96%
macro/parse_log	100	0.309	**0.278**	11.17%
macro/pi	1000	**0.024**	0.027	12.38%
macro/pi	10000	**2.034**	2.444	20.19%
macro/rcs	100	**0.581**	0.634	9.26%
macro/sudoku	1	**1.661**	1.859	11.91%
micro/app_factorial	5000	**0.037**	0.047	25.63%
micro/app_fib	30	**0.189**	0.192	1.97%
micro/app_fib	35	**2.099**	2.122	1.10%
micro/app_mandelbrot	1	**0.277**	0.288	3.79%
micro/app_pentomino	1	**24.784**	25.306	2.11%
micro/app_tak	7	0.143	**0.140**	2.45%
micro/app_tak	8	**0.414**	0.401	3.19%
micro/app_tak	9	1.091	**1.056**	3.36%
micro/app_tarai	3	0.492	**0.487**	1.06%
micro/app_tarai	4	0.604	**0.589**	2.60%
micro/app_tarai	5	0.731	**0.713**	2.53%
micro/binary_trees	1	**12.500**	12.678	1.43%
micro/count_multithreaded	1	**0.006**	0.007	12.19%
micro/count_multithreaded	2	**0.012**	0.014	13.36%
micro/count_multithreaded	4	**0.025**	0.027	10.67%
micro/count_multithreaded	8	**0.049**	0.055	11.09%
micro/count_multithreaded	16	**0.100**	0.110	10.42%
micro/count_shared_thread	1	**0.061**	0.067	9.37%
micro/count_shared_thread	2	**0.061**	0.068	11.06%
micro/count_shared_thread	4	**0.061**	0.067	10.12%
micro/count_shared_thread	8	**0.062**	0.069	10.24%
micro/count_shared_thread	16	**0.062**	0.069	11.56%
micro/eval	1000000	**5.892**	6.408	8.76%
micro/fannkuch	6	**0.003**	0.004	16.37%
micro/fannkuch	8	**0.252**	0.291	15.35%
micro/fannkuch	10	**29.300**	33.422	14.07%
micro/fasta	1000000	**13.992**	14.939	6.77%
micro/fiber_ring	10	**0.000**	0.000	13.89%
micro/fiber_ring	100	0.018	**0.017**	8.72%
micro/fiber_ring	1000	**1.724**	1.948	13.04%
micro/fractal	5	**3.004**	3.081	2.55%
micro/gc_array	1	**39.619**	100.169	152.83%
micro/gc_mb	500000	**0.173**	0.212	22.83%
micro/gc_mb	1000000	**0.346**	0.412	19.09%

Table 5.11 — continued on the next page

Table 5.11 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1 (FreeBSD)	Ratio
micro/gc_mb	3000000	**1.087**	1.424	31.05%
micro/gc_string	1	**3.090**	4.339	40.42%
micro/knucleotide	1	**0.826**	0.854	3.37%
micro/lucas_lehmer	9689	4.611	**4.498**	2.53%
micro/lucas_lehmer	9941	4.980	**4.852**	2.63%
micro/lucas_lehmer	11213	7.096	**6.924**	2.49%
micro/lucas_lehmer	19937	38.069	**36.967**	2.98%
micro/mandelbrot	1	30.207	**29.574**	2.14%
micro/mbari_bogus1	1	**0.008**	0.012	54.43%
micro/mergesort	1	**0.659**	0.697	5.79%
micro/mergesort_hongli	3000	**1.108**	1.150	3.85%
micro/meteor_contest	1	**7.633**	7.825	2.51%
micro/monte_carlo_pi	10000000	**7.828**	7.924	1.24%
micro/nbody	100000	**5.569**	6.258	12.38%
micro/nsieve	9	NoMethodError	NoMethodError	
micro/nsieve_bits	8	**2.385**	2.499	4.79%
micro/open_many_files	50000	**0.245**	0.433	77.22%
micro/partial_sums	2500000	**14.396**	16.611	15.39%
micro/primes	3000	**0.012**	0.013	9.31%
micro/primes	30000	0.128	**0.126**	1.87%
micro/primes	300000	1.404	**1.367**	2.71%
micro/primes	3000000	17.761	**17.544**	1.24%
micro/quicksort	1	**1.432**	1.501	4.81%
micro/read_large	100	**2.441**	2.881	18.03%
micro/regex_dna	20	2.797	**2.760**	1.34%
micro/reverse_compliment	1	**2.782**	3.344	20.22%
micro/simple_connect	1	0.216	SocketError	
micro/simple_connect	100	0.143	SocketError	
micro/simple_connect	500	0.173	SocketError	
micro/simple_server	1	0.138	SocketError	
micro/simple_server	100	0.138	SocketError	
micro/simple_server	100000	1.411	SocketError	
micro/so_ackermann	7	0.059	**0.056**	4.98%
micro/so_ackermann	9	0.957	**0.909**	5.24%
micro/so_array	9000	**1.938**	2.057	6.11%
micro/so_count_words	100	**2.831**	3.453	21.98%
micro/so_exception	500000	**8.324**	9.018	8.33%
micro/so_lists	1000	**4.483**	4.876	8.76%
micro/so_lists_small	1000	**0.905**	0.985	8.78%
micro/so_matrix	60	0.592	**0.554**	6.95%
micro/so_object	500000	**0.413**	0.431	4.28%
micro/so_object	1000000	**0.817**	0.862	5.52%

Table 5.11 — continued on the next page

Table 5.11 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1 (FreeBSD)	Ratio
micro/so_object	1500000	**1.224**	1.294	5.67%
micro/so_sieve	4000	**8.650**	9.031	4.40%
micro/socket_transfer_1mb	10000	0.306	SocketError	
micro/socket_transfer_1mb	1000000	0.304	SocketError	
micro/spectral_norm	100	0.233	**0.230**	1.00%
micro/string_concat	10000000	**1.525**	2.793	83.09%
micro/sum_file	100	**3.817**	4.081	6.94%
micro/word_anagrams	1	**3.775**	4.054	7.40%
micro/write_large	100	**0.147**	0.470	219.43%
rdoc/against_itself_darkfish	1	**6.991**	7.561	8.16%
rdoc/against_itself_ri	1	**5.580**	9.178	64.47%
rdoc/core_darkfish	1	SystemExit	114.611	
Sum		**369.389**	**451.437**	**22.21%**

Table 5.11 shows the results of the YARV benchmark. Similarly to MRI's benchmark, YARV has a better overall performance in Linux. The whole benchmark took 22.21% more time to complete under FreeBSD. The most prominent results are equally I/O related. Under FreeBSD, garbage collecting an array in YARV is slower than doing it in MRI, which is an unexpected outcome. This activity is approximately 2,5 times faster when performed under Linux.

Somewhat expectedly, YARV and MRI's performance is considerably better in the Linux operating system. After eliminating FreeBSD from the benchmarking subjects, Gentoo is the Linux distribution that will be used in future work. It is very stable, configurable and enables improved performance of Ruby-related software when compared to FreeBSD.

5.1.3 Tweaking

There are many configurations and options that can be fine-tuned in operating systems. Sysctl enables kernel parameter configuration at runtime. The aforeshown web server benchmarks required some optimization changes to improve the system's stability under such high-load. These are shown on table 5.12. Options 1, 2, 3 and 4 increase the TCP buffers on read/write, improving the system performance when dealing with big transfers. Options 5 and 6 increase the number of connections which are allowed to be queued behind a busy kernel. Options 7 and 8 enable socket reusing and fast socket recycling. Option 9 decreases the time allowed for a socket to exist without a connection. Option 10 disables timestamps in packet headers, reducing the packet's size. Finally, option 11 decreases the number of failed retries before killing the TCP connection.

The number of opened files limit also had to be increased in the system's limits configuration. It generally defaults to 1024 which is appropriate for desktop systems but very low on a server environment. Taking into account that each socket connection uses a file on a UNIX system, the

Table 5.12: Sysctl Options and Values

#	NAME	VALUE
1	net.core.rmem_max	16777216
2	net.core.wmem_max	16777216
3	net.ipv4.tcp_rmem	4096 87380 16777216
4	net.ipv4.tcp_wmem	4096 87380 16777216
5	net.core.netdev_max_backlog	4096
6	net.core.somaxconn	4096
7	net.ipv4.tcp_tw_reuse	1
8	net.ipv4.tcp_tw_recycle	1
9	net.ipv4.tcp_fin_timeout	15
10	net.ipv4.tcp_timestamps	0
11	net.ipv4.tcp_orphan_retries	1

default setting would generally cap the system's concurrency ability to $\pm$1000 requests, so it was increased to 65536.

A few other options are worth investigating. Many server-oriented distributions use the Deadline I/O scheduler which gives a higher priority to read requests, while others use the fair CFQ I/O scheduler which has balanced priorities and is commonly found in desktop systems. Preemption should also be disabled on a server kernel. In non-preemptive configurations, kernel code runs until completion—the scheduler can not touch it until it is finished. Server kernels should also have their timer interrupt rate set to 100Hz, which causes higher latency but lower overhead, yielding superior raw processing power.

All the aforementioned configuration changes were in use in during all benchmarks.

5.1.4 Section Overview

This section exhibited and explained the work concerning operating systems. Regarding development, a general benchmarking script oriented towards UNIX systems was created. On the benchmarking phase, this script was used on four Linux distributions—Debian, Ubuntu Server, CentOS and Gentoo—where CentOS underperformed and was discarded from future work. Then, a benchmark focused on web server performance was made using the three remaining Linux distributions. Gentoo yielded the best performance while maintaining a remarkable stability. It was then compared with FreeBSD, by using a renowned Ruby benchmarking suite. Having overall better performance, Gentoo was chosen to be the base for all future work. Finally, concerning tweaking, a few important kernel options were presented and their impact on performance was explained.

The results obtained when analyzing general, web server and Ruby performance allowed to complete the general goal initially stated. Given its results, Gentoo poses as an excellent operating system to be paired with a Rails setup.

5.2 Ruby

Concerning this component, the main focus was divided into four core activities. First of all, determining the real benefits of upgrading to the latest Ruby 1.9. After that, focus would shift towards porting *Escolinhas.pt* to Ruby 1.9. Then, the aim was to improve YARV's GC by increasing its flexibility. Finally, Ruby's profiling and information retrieval capabilities were enhanced.

There are many interpreters being used in production environments whose characteristics were explored in Section 2.2. Unfortunately, none except YARV fully support the most recent specification—1.9. It is likely that focusing on soon to be outdated solutions is not worthy. For this reason, YARV was the interpreter targeted for improvements.

Determining the performance benefits from upgrading to the most recent version of Ruby, analyzing the effort needed to accomplish it, improving one of YARV's main bottlenecks related with Rails—the garbage collector—and greatly enhancing its profiling capabilities were the general goals of the work presented in this section.

5.2.1 Benchmarking

As mentioned in Section 3.2 the new Ruby interpreter, YARV, is supposed to significantly improve performance over its predecessor, MRI. Its first release happened more than two years ago and its adoption is still very low, despite having new features and supposedly better performance. Porting existing applications requires some development effort as there are small changes on existing functionality and behavior. Having said that Rails can easily have scalability issues, it would be expected that this version's adoption would rise considerably fast but, however, this is not currently happening.

To increase the community's awareness of the benefits of upgrading, these need to be accurately determined. The Ruby benchmark mentioned and explained on Section 5.1 was rerun to exhibit the interpreter's disparity when it comes to performance, as shown on table 5.13.

Table 5.13: MRI and YARV Benchmark Comparison

Benchmark	Input Size	MRI (1.8.7)	YARV (1.9.1)	Ratio
macro/cal	500	1.990	**0.289**	589.28%
macro/dirp	10000	**0.386**	0.393	2.04%
macro/gzip	100	6.141	**5.979**	2.70%
macro/hilbert_matrix	10	0.036	**0.002**	1509.42%
macro/hilbert_matrix	20	0.335	**0.031**	984.21%
macro/hilbert_matrix	30	1.367	**0.154**	789.44%
macro/hilbert_matrix	40	3.866	**0.477**	710.68%
macro/hilbert_matrix	50	8.868	**1.256**	605.76%
macro/hilbert_matrix	60	18.399	**3.024**	508.39%
macro/list	1000	0.053	**0.026**	102.35%
macro/list	10000	7.154	**2.758**	159.43%

Table 5.13 — continued on the next page

Table 5.13 — continued from previous page

Benchmark	Input Size	MRI (1.8.7)	YARV (1.9.1)	Ratio
macro/mpart	300	0.039	**0.034**	16.56%
macro/norvig_spelling	50	8.562	ArgumentError	
macro/observ	100000	0.612	**0.360**	69.88%
macro/parse_log	100	1.151	**0.309**	271.94%
macro/pi	1000	0.026	**0.024**	7.46%
macro/pi	10000	2.127	**2.034**	4.61%
macro/rcs	100	0.753	**0.581**	29.70%
macro/sudoku	1	10.153	**1.661**	511.27%
micro/app_factorial	5000	StackError	0.037	
micro/app_fib	30	1.587	**0.189**	741.29%
micro/app_fib	35	17.854	**2.099**	750.68%
micro/app_mandelbrot	1	1.899	**0.277**	585.27%
micro/app_pentomino	1	SignalException	24.784	
micro/app_tak	7	1.173	**0.143**	719.77%
micro/app_tak	8	3.405	**0.414**	722.62%
micro/app_tak	9	8.910	**1.091**	716.51%
micro/app_tarai	3	3.873	**0.492**	686.81%
micro/app_tarai	4	4.670	**0.604**	673.22%
micro/app_tarai	5	5.654	**0.731**	672.95%
micro/binary_trees	1	54.757	**12.500**	338.06%
micro/count_multithreaded	1	**0.004**	0.006	40.88%
micro/count_multithreaded	2	**0.009**	0.012	41.65%
micro/count_multithreaded	4	**0.017**	0.025	45.27%
micro/count_multithreaded	8	**0.034**	0.049	46.36%
micro/count_multithreaded	16	**0.067**	0.100	48.25%
micro/count_shared_thread	1	**0.044**	0.061	39.60%
micro/count_shared_thread	2	**0.044**	0.061	39.45%
micro/count_shared_thread	4	**0.044**	0.061	38.36%
micro/count_shared_thread	8	**0.044**	0.062	40.22%
micro/count_shared_thread	16	**0.045**	0.062	38.67%
micro/eval	1000000	**1.843**	5.892	219.67%
micro/fannkuch	6	0.005	**0.003**	54.48%
micro/fannkuch	8	0.398	**0.252**	57.81%
micro/fannkuch	10	44.995	**29.300**	53.57%
micro/fasta	1000000	37.028	**13.992**	164.65%
micro/fiber_ring	10	LoadError	0.000	
micro/fiber_ring	100	LoadError	0.018	
micro/fiber_ring	1000	LoadError	1.724	
micro/fractal	5	4.520	**3.004**	50.48%
micro/gc_array	1	44.990	**39.619**	13.56%
micro/gc_mb	500000	0.687	**0.173**	297.95%
micro/gc_mb	1000000	1.513	**0.346**	337.27%

Table 5.13 — continued on the next page

Table 5.13 — continued from previous page

Benchmark	Input Size	MRI (1.8.7)	YARV (1.9.1)	Ratio
micro/gc_mb	3000000	3.535	**1.087**	225.32%
micro/gc_string	1	7.874	**3.090**	154.80%
micro/knucleotide	1	1.396	**0.826**	68.93%
micro/lucas_lehmer	9689	**4.019**	4.611	14.73%
micro/lucas_lehmer	9941	**4.342**	4.980	14.68%
micro/lucas_lehmer	11213	**6.174**	7.096	14.93%
micro/lucas_lehmer	19937	**33.143**	38.069	14.87%
micro/mandelbrot	1	55.460	**30.207**	83.60%
micro/mbari_bogus1	1	StackError	0.008	
micro/mergesort	1	1.914	**0.659**	190.40%
micro/mergesort_hongli	3000	4.423	**1.108**	299.34%
micro/meteor_contest	1	25.352	**7.633**	232.14%
micro/monte_carlo_pi	10000000	12.178	**7.828**	55.58%
micro/nbody	100000	7.046	**5.569**	26.53%
micro/nsieve	9	15.221	NoMethodError	
micro/nsieve_bits	8	19.753	**2.385**	728.28%
micro/open_many_files	50000	**0.211**	0.245	16.05%
micro/partial_sums	2500000	18.003	**14.396**	25.05%
micro/primes	3000	5.057	**0.012**	43396.39%
micro/primes	30000	SignalException	0.128	
micro/primes	300000	SignalException	1.404	
micro/primes	3000000	SignalException	17.761	
micro/quicksort	1	7.116	**1.432**	396.99%
micro/read_large	100	6.337	**2.441**	159.65%
micro/regex_dna	20	**2.644**	2.797	5.78%
micro/reverse_compliment	1	3.702	**2.782**	33.08%
micro/simple_connect	1	**0.134**	0.216	60.82%
micro/simple_connect	100	**0.143**	0.143	0.40%
micro/simple_connect	500	0.175	**0.173**	0.72%
micro/simple_server	1	**0.134**	0.138	3.02%
micro/simple_server	100	**0.136**	0.138	1.89%
micro/simple_server	100000	1.427	**1.411**	1.12%
micro/so_ackermann	7	0.494	**0.059**	734.56%
micro/so_ackermann	9	StackError	0.957	
micro/so_array	9000	6.867	**1.938**	254.26%
micro/so_count_words	100	**2.408**	2.831	17.52%
micro/so_exception	500000	**8.009**	8.324	3.93%
micro/so_lists	1000	8.649	**4.483**	92.91%
micro/so_lists_small	1000	1.741	**0.905**	92.30%
micro/so_matrix	60	1.924	**0.592**	224.80%
micro/so_object	500000	1.424	**0.413**	244.44%
micro/so_object	1000000	2.825	**0.817**	245.87%

Table 5.13 — continued on the next page

Table 5.13 — continued from previous page

Benchmark	Input Size	MRI (1.8.7)	YARV (1.9.1)	Ratio
micro/so_object	1500000	4.265	**1.224**	248.29%
micro/so_sieve	4000	54.784	**8.650**	533.31%
micro/socket_transfer_1mb	10000	0.353	**0.306**	15.40%
micro/socket_transfer_1mb	1000000	0.356	**0.304**	17.28%
micro/spectral_norm	100	0.932	**0.233**	300.57%
micro/string_concat	10000000	5.655	**1.525**	270.78%
micro/sum_file	100	9.920	**3.817**	159.90%
micro/word_anagrams	1	7.750	**3.775**	105.31%
micro/write_large	100	0.157	**0.147**	6.51%
rdoc/against_itself_darkfish	1	13.118	**6.991**	87.65%
rdoc/against_itself_ri	1	12.854	**5.580**	130.35%
Total		**679.880**	**325.398**	**108.94%**

As expected, YARV shows excellent results on this benchmark, in comparison with MRI. It is approximately 109% faster than the older 1.8 interpreter. Concerning memory usage, MRI used an average 46,34MB of memory during the whole benchmark while YARV only consumed 30,81MB. Regarding specific tests, YARV is generally faster by there is a notable exception: threads. As explained in Section 2.2, MRI uses green threads while YARV supports native threads. Green threads emulate multithreaded environments but there is no parallel computing at all since they do not use multiple processing units. They have, however, a significantly low spawning overhead, contrary to native threads. The previously shown threading tests, where MRI yielded better results than YARV, are very fast. For this reason, the overhead of spawning native threads for such a small test duration had a considerable impact in the overall performance of YARV, resulting in poorer performance. Heavier tests would likely yield different results, given that native threads are actually capable of running on different processing units.

Notwithstanding this exception, YARV shows significant performance improvements overall and this fact is likely to motivate developers to switch to this version.

It is also critical to determine the performance differences on a real Rails application. As this is also related to the web server in use, this subject will be researched throughly on Section 5.3.

5.2.2 Development

The development phase involved many distinct activities. The details on each one of them are explored and explained below.

5.2.2.1 Porting Escolinhas.pt to Ruby 1.9

Escolinhas.pt has over 70 models and database tables, over 130000 lines of Ruby code, uses over 40 Rails plugins and gems, and there are currently more than 10 development branches. It is

a heavy and complex full-featured application, making it a great subject to evaluate the effort needed to port an application to Ruby 1.9.

This process was fairly simple. The main issue was related to character encodings in Portuguese literal strings on the source code and database, which emerged from the heavy changes regarding encoding handling in Ruby 1.9. A patch to fix literal string encoding can be found on appendix A. Ruby 1.9 also requires developers to set the default encoding for each file inside a comment in the beginning, or else it will use the default ASCII-8BIT which has its known limitations. A task was developed to manage the default encoding in all files of a Ruby project and is presented on appendix B.

Other issues were mainly related to extracted functionality and syntax changes. The most common ones were:

- *Object.type* changed to *Object.class.name*;
- The *String* class no longer has the *normalize* method;
- The case statement no longer supports ":" to separate the match word from the action to be taken.

As mentioned before, porting *Escolinhas.pt* to Ruby 1.9 was a very fast and simple process. Measuring the effort in time units, porting to 1.9 accounted for less then 0.1% of all the development endeavor. All changes were simple and straightforward. Taking the aforementioned performance benefits into account, porting an application to Ruby 1.9 is likely to be very worthy on most projects.

On a related matter, a considerable amount of effort is being made by the Ruby on Rails development team to automatically manage the encoding of Ruby files in Rails 3. Given that Ruby's encoding changes accounted for most part of the effort needed to port *Escolinhas.pt* to Ruby 1.9, porting an application under Rails 3 should generally be effortless, as this version natively handles most of Ruby's encoding-related quirks.

5.2.2.2 Increasing YARV's GC Flexibility

Ruby Enterprise Edition, as exposed in Section 2.2, allows users to set some GC parameters, providing adaptive performance. As mentioned in Section 3.2, many platforms benefit from this flexibility by adapting many GC parameters to their applications. Adding this functionality to YARV was the goal, and it currently supports the following settings:

RUBY_HEAP_MIN_SLOTS, the initial number of heap slots. It also represents the minimum number of slots at all times (default: 10000);

RUBY_HEAP_SLOTS_INCREMENT, the number of new slots to allocate when all initial slots are used (default: 10000);

RUBY_HEAP_SLOTS_GROWTH_FACTOR, the multiplicator used next time Ruby needs new heap slots (default: 1.8, meaning it will allocate 18000 new slots if the default settings are in use);

RUBY_GC_MALLOC_LIMIT, the number of C data structures that can be allocated before triggering the garbage collector. This one is very important since the default value makes the GC run when there are still empty heap slots, mainly due to Rails frequently allocating and deallocating large amounts of data (default: 8000000);

RUBY_HEAP_FREE_MIN, the number of free slots that should be present after GC finishes running. In case there are fewer slots than those defined, it will allocate new ones according to value of *RUBY_HEAP_SLOTS_INCREMENT* and the value of the previously mentioned *RUBY_HEAP_SLOTS_GROWTH_FACTOR* parameters (default: 4096).

The developer can configure these options in the system's environment and the Ruby interpreter will detect and apply them to its internal configuration.

Appendix C contains a sample script which launches Ruby with a user-defined configuration. The parameters indicate that Ruby should start with more initial slots, have a higher threshold for triggering the GC and have a less exponential memory growth—1,1—instead of using the default 1,8 coefficient.

A benchmark comparing these configurations with the default ones was conducted and is presented on table 5.14.

Table 5.14: Flexible YARV Benchmark

Benchmark	Input Size	YARV	YARV (tweaked)	Ratio
macro/cal	500	0.289	**0.253**	14.03%
macro/dirp	10000	**0.393**	0.454	15.38%
macro/gzip	100	**5.979**	5.991	0.20%
macro/hilbert_matrix	10	0.002	**0.002**	12.81%
macro/hilbert_matrix	20	0.031	**0.027**	13.03%
macro/hilbert_matrix	30	0.154	**0.133**	15.65%
macro/hilbert_matrix	40	0.477	**0.399**	19.47%
macro/hilbert_matrix	50	1.256	**1.022**	22.95%
macro/hilbert_matrix	60	3.024	**2.296**	31.72%
macro/list	1000	0.026	**0.025**	7.04%
macro/list	10000	2.758	**1.418**	94.42%
macro/mpart	300	0.034	**0.032**	4.63%
macro/norvig_spelling	50	ArgumentError	4.781	
macro/observ	100000	0.360	**0.336**	7.14%
macro/parse_log	100	**0.309**	0.460	48.72%
macro/pi	1000	0.024	**0.023**	3.31%
macro/pi	10000	2.034	**1.979**	2.77%

Table 5.14 — continued on the next page

Table 5.14 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1(FreeBSD)	Ratio
macro/rcs	100	0.581	**0.491**	18.16%
macro/sudoku	1	1.661	**1.570**	5.77%
micro/app_factorial	5000	0.037	**0.036**	3.83%
micro/app_fib	30	0.189	**0.177**	6.47%
micro/app_fib	35	2.099	**1.954**	7.43%
micro/app_mandelbrot	1	0.277	**0.242**	14.64%
micro/app_pentomino	1	24.784	**23.880**	3.79%
micro/app_tak	7	0.143	**0.138**	3.93%
micro/app_tak	8	0.414	**0.395**	4.72%
micro/app_tak	9	1.091	**1.054**	3.58%
micro/app_tarai	3	0.492	**0.485**	1.54%
micro/app_tarai	4	0.604	**0.585**	3.31%
micro/app_tarai	5	0.731	**0.708**	3.27%
micro/binary_trees	1	12.500	**11.870**	5.30%
micro/count_multithreaded	1	0.006	**0.006**	3.11%
micro/count_multithreaded	2	0.012	**0.012**	3.74%
micro/count_multithreaded	4	0.025	**0.024**	4.68%
micro/count_multithreaded	8	0.049	**0.047**	4.26%
micro/count_multithreaded	16	0.100	**0.095**	4.93%
micro/count_shared_thread	1	0.061	**0.059**	3.22%
micro/count_shared_thread	2	0.061	**0.059**	3.06%
micro/count_shared_thread	4	0.061	**0.059**	3.30%
micro/count_shared_thread	8	0.062	**0.059**	5.18%
micro/count_shared_thread	16	0.062	**0.059**	4.90%
micro/eval	1000000	**5.892**	6.630	12.53%
micro/fannkuch	6	0.003	**0.003**	8.08%
micro/fannkuch	8	0.252	**0.241**	4.60%
micro/fannkuch	10	**29.300**	29.922	2.12%
micro/fasta	1000000	13.992	**12.560**	11.40%
micro/fiber_ring	10	0.000	**0.000**	2.96%
micro/fiber_ring	100	0.018	**0.018**	2.59%
micro/fiber_ring	1000	1.724	**1.682**	2.47%
micro/fractal	5	3.004	**2.326**	29.12%
micro/gc_array	1	39.619	**15.785**	150.99%
micro/gc_mb	500000	**0.173**	0.233	35.07%
micro/gc_mb	1000000	**0.346**	0.470	35.84%
micro/gc_mb	3000000	**1.087**	1.400	28.84%
micro/gc_string	1	3.090	**2.857**	8.16%
micro/knucleotide	1	0.826	**0.737**	12.09%
micro/lucas_lehmer	9689	4.611	**4.436**	3.96%
micro/lucas_lehmer	9941	4.980	**4.792**	3.92%
micro/lucas_lehmer	11213	7.096	**6.830**	3.89%

Table 5.14 — continued on the next page

Table 5.14 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1(FreeBSD)	Ratio
micro/lucas_lehmer	19937	38.069	**36.632**	3.92%
micro/mandelbrot	1	30.207	**22.877**	32.04%
micro/mbari_bogus1	1	**0.008**	0.008	4.31%
micro/mergesort	1	0.659	**0.621**	6.16%
micro/mergesort_hongli	3000	1.108	**1.014**	9.26%
micro/meteor_contest	1	7.633	**6.275**	21.64%
micro/monte_carlo_pi	10000000	7.828	**6.126**	27.78%
micro/nbody	100000	5.569	**4.634**	20.18%
micro/nsieve_bits	8	2.385	**2.087**	14.28%
micro/open_many_files	50000	0.245	**0.214**	14.53%
micro/partial_sums	2500000	14.396	**12.077**	19.20%
micro/primes	3000	0.012	**0.010**	16.46%
micro/primes	30000	0.128	**0.113**	12.88%
micro/primes	300000	1.404	**1.208**	16.22%
micro/primes	3000000	17.761	**13.421**	32.34%
micro/quicksort	1	1.432	**1.429**	0.21%
micro/read_large	100	2.441	**2.078**	17.47%
micro/regex_dna	20	**2.797**	3.082	10.21%
micro/reverse_compliment	1	2.782	**2.741**	1.48%
micro/simple_connect	1	0.216	**0.132**	63.98%
micro/simple_connect	100	0.143	**0.138**	4.10%
micro/simple_connect	500	0.173	**0.166**	4.26%
micro/simple_server	1	0.138	**0.131**	5.25%
micro/simple_server	100	0.138	**0.132**	4.77%
micro/simple_server	100000	1.411	**1.322**	6.72%
micro/so_ackermann	7	0.059	**0.052**	12.89%
micro/so_ackermann	9	0.957	**0.845**	13.28%
micro/so_array	9000	1.938	**1.832**	5.78%
micro/so_count_words	100	**2.831**	2.877	1.64%
micro/so_exception	500000	**8.324**	9.251	11.13%
micro/so_lists	1000	4.483	**4.383**	2.28%
micro/so_lists_small	1000	0.905	**0.884**	2.45%
micro/so_matrix	60	0.592	**0.553**	7.09%
micro/so_object	500000	0.413	**0.358**	15.40%
micro/so_object	1000000	0.817	**0.722**	13.15%
micro/so_object	1500000	1.224	**1.074**	13.97%
micro/so_sieve	4000	**8.650**	8.710	0.69%
micro/socket_transfer_1mb	10000	**0.306**	0.391	27.95%
micro/socket_transfer_1mb	1000000	**0.304**	0.396	30.21%
micro/spectral_norm	100	0.233	**0.193**	20.60%
micro/string_concat	10000000	**1.525**	1.592	4.38%
micro/sum_file	100	3.817	**3.483**	9.59%

Table 5.14 — continued on the next page

Table 5.14 — continued from previous page

Benchmark	Input Size	1.9.1 (Gentoo)	1.9.1(FreeBSD)	Ratio
micro/word_anagrams	1	**3.775**	3.940	4.37%
micro/write_large	100	0.147	**0.140**	4.95%
rdoc/against_itself_darkfish	1	**6.991**	7.199	2.97%
rdoc/against_itself_ri	1	**5.580**	5.653	1.30%
rdoc/core_darkfish	1	SystemExit	66.191	
Total		**372.219**	**323.032**	**15.23%**

The aforementioned parameters changed YARV's performance, having an overall improvement of ±15,2%. Since it was configured to have a less sensible threshold, garbage collection was triggered less often resulting in improved execution times. The usage of these parameters also slightly improved the average memory usage throughout all tests, lowering it from 30,81MB to 29,09MB. This is mainly due to the fact that the allocation rate is less exponential thus provoking a more linear heap growth.

The Ruby fork where this additional feature was implemented is open-source and hosted on GitHub [10].

5.2.2.3 Storage and Retrieval of Profiling Information

The most recent version of Ruby already possesses an embedded profiler. It records important information like the time spent in garbage collection, the number of live objects in the heap or even the total size of allocated memory, among others. Users can retrieve its current status in a properly formatted string containing all the data.

This current implementation has three main issues that can negatively impact many profiling situations. Firstly, "string" being the only format in which this information can be retrieved. Though being optimized for the Human eye, it is not very suited for automated processing. The second issue is related with the profiler's inability to record data accumulatively. For instance, if a given task allocates one thousand objects but triggers the GC a number of times, the developer will not be aware of those one thousand object allocations since the garbage collector will have probably freed some of them. Lastly, the profiler does not track down the number of times that the garbage collector is triggered. This information is crucial to optimize applications or code snippets since, based on what was explained in Section 3.2, fine-tuning an application to trigger the garbage collector less often will increase the interpreter's performance.

Given these issues, the development phase had three clear goals:

1. Add the ability to retrieve profiling information in hash format;
2. Record the accumulated number and size of data allocations;
3. Record the number of times that the garbage collector is triggered.

[10] http://github.com/goncalossilva/ruby/

This work was developed as a series of open-source patches for the most recent Ruby versions, stable or unstable, and is hosted on GitHub [11].

5.2.2.4 Graphical Interface for Profiling Applications

There are many tools available to profile Ruby and Rails applications. However, most of them record their analysis information in text. While this format suits most situations as very convenient, a graphical interface would be more appropriate in some cases. A solid example of this situation is a call stack. Albeit being acceptable to read and understand a small call stack in a console or text file, Rails applications tend to have huge call stacks which, unfortunately, are very hard to read and understand on this format.

Stephen Kaes once developed an hierarchical HTML output format for ruby-prof [12] which is one of the most famous Ruby profilers and the only one that is officially supported in Rails. However, its development was halted and as the development of Ruby advanced this HTML printer soon became deprecated and stopped working. Users on the most recent versions of Ruby, from 1.8.7 to 1.9.2, could not use this printer to analyze the call stacks of their applications or code snippets.

The goal was to get this hierarchical HTML printer working with the latest Ruby versions. Firstly, it involved getting ruby-prof's compatibility with Ruby 1.9 increased and then porting the printer to this up-to-date version.

This work was done in collaboration with Yehuda Katz [13], one of Rails' core team members. The working printer [14] is part of my public "ruby-prof" branch on GitHub [15].

It hierarchically shows the call stack, being able to expand or collapse calls. It shows the total amount of time of the test, the percentage each call represents of the complete execution time and the number of times each method was called. If multiple threads ran different calls, it represents them accordingly. A percentage threshold can also be defined in order to collapse faster calls and highlight the slower ones. Finally, it uses colors to indicate each call's speed, improving bottleneck detection. An example output is shown on appendix D.

5.2.3 Section Overview

This section exhibited and explained the work concerning the Ruby interpreter. Regarding benchmarking, the performance differences between YARV and its predecessor, MRI, were determined. As expected, YARV overall performance is noticeably better than MRI's. Concerning development, many activities were involved. First of all, *Escolinhas.pt* was ported to Ruby 1.9 to benefit from YARV's improvements. The porting effort was analyzed, concluding that it was significantly

[11] http://github.com/wycats/ruby-prof/tree/master/patches/gcdata/
[12] http://ruby-prof.rubyforge.org/
[13] http://yehudakatz.com/
[14] http://github.com/wycats/ruby-prof/blob/master/lib/ruby-prof/call_stack_printer.rb
[15] http://github.com/wycats/ruby-prof/

low. After that, YARV's GC flexibility was improved by creating five configuration options for adaptive performance. Initial tests were shown and by using the settings mentioned in appendix C YARV's performance increased while consuming less memory overall. YARV's internal profiler was also improved. More data is being recorded, the *hash* format is now supported on result output and an HTML-based output format was incorporated, providing a graphical interface for easier analysis.

All benchmarks comparing MRI with YARV allowed determining the overall benefits of upgrading applications to YARV. Porting *Escolinhas.pt* to Ruby 1.9 allowed to briefly analyze the effort needed to upgrade applications to Ruby 1.9. Adding flexibility to YARV and analyzing its performance under different configurations allowed concluding that YARV's garbage collector was improved. By recording more data and providing alternate output formats, YARV's profiling capabilities were also enhanced, completing the last of the general goals initially stated.

5.3 Rails Web Servers

There are many Rails-compatible web servers with distinct philosophies and purposes. Passenger, Thin and Unicorn were targeted for benchmarking, in an effort to determine which is best for each situation. Memory usage and stability are important factors and were taken into account when evaluating each alternative. Apache, Nginx and Cherooke's performance as reverse proxies was also analyzed.

Among the previously introduced web servers, some are missing from this analysis. Ruby's pioneer web server—WEBrick—is left out since it lacks the efficiency to compete with nowadays alternatives. Mongrel, while representing a huge improvement over WEBrick, still lags behind more recently developed web servers as explored in Section 3.3, also being excluded from further analysis.

Throughly analyzing the performance, stability and memory usage of up-to-date web server setups was the general goal of the work presented in this section.

5.3.1 Development

Monitoring memory usage is crucial to evaluate the performance of a web server. A web server that consumes less memory is likely to scale better since it can spawn more working processes/threads in the same setup.

There was need for a small utility to measure the memory usage of specified processes over time, providing the ability to configure its refresh interval. It had to be very light not to interfere with the tests, should rely on the operating system's data and preferably output in a easily parsable format. This tool was created and is presented on appendix E. It takes a few arguments, namely:

- Directory to use when outputting the results;
- Refresh interval in seconds (optional, default: 1);

- Name of the process(es) to monitor.

The utility records each process's memory usage in its own file. For each process, it will record the data in a file entitled *process_name.process_pid.csv*. As the name implies, it records the data in the CSV format. Finally, it is compatible with any UNIX system.

This tool was used to measure the memory usage in all benchmarks contained in the following section.

5.3.2 Benchmarking

Ruby has a few highly performant web servers with Passenger, Thin and Unicorn among them. Exchanging web servers in a Rails' setup is not an uncommon activity since improved versions are being frequently released in recent years, as stated in Section 2.3.

Regarding this component, benchmarking had four distinct phases. First of all, it was important to evaluate the performance of Apache, Nginx and Cherokee as reverse proxies. After that, the goal was to determine whether Passenger performs better in combination with Apache or Nginx. The third phase would compare Thin's performance to Unicorn's since, as stated on Section 2.3, they are developed for similar purposes. All these initial phases would use MRI as the Ruby interpreter. Finally, a more complete benchmark including Passenger, Thin and Unicorn was done, involving multiple quantities of workers and running with MRI and YARV.

For all benchmarks, three distinct pages were used. One *heavy page* filled with dynamic content, a *regular page* with moderate usage of dynamic content and finally a complex but small *API call*. *Escolinhas.pt* was the platform to provide this different types of content.

No database caching solutions were in use. Once again, if more than 1% of the requests took more than 30 seconds to be replied to the test was considered a failure, as it already surpasses acceptable response times in real world applications.

5.3.2.1 Proxy Performance in a Thin Cluster Environment

Many Ruby web servers run behind a reverse proxy which buffers the requests, delivers them to the web server and receives the replies, buffering them as well if needed. In this kind of commonly used architecture, they have an important role when it comes to performance.

Thin is a kind of web server that should be paired with a reverse proxy. It is optimized for small requests and fast clients so it needs to rely on the proxy's buffering abilities to offer consistent performance across all kinds of light or heavy content and fast or slow clients.

This benchmark used 4 Thin processes. The only modifications made to its default configuration were related to increasing its request queue, so that it could queue more clients instead of discarding them. This Thin cluster was working behind the 3 aforementioned reverse proxies—Apache, Nginx and Cherooke. The benchmark involved many levels of concurrency, as follows:

- 50 requests, 1 concurrent;
- 100 requests, 10 concurrent;

- 500 requests, 50 concurrent;
- 500 requests, 100 concurrent;
- 2500 requests, 500 concurrent.

The total time to complete each test and the replies *per second* capability of each setup were recorded using the aforementioned tool—*ab*—created by the Apache Foundation[16]. When using this tool, the user specifies the total amount of requests to be sent and the desired concurrency. It then dispatches all requests to the specified host at the chosen concurrency rate.

The results of this benchmark are presented on table 5.15.

Table 5.15: Reverse Proxy Benchmark

Req. / Conc.	Page	Proxy	Requests/s (#)	Total time (s)	Mem. Usage (B)
50/1	API	Cherokee	**9.96**	**5.02**	167553
		Apache	9.88	5.058	200126
		Nginx	9.24	5.411	**121213**
	Heavy	Cherokee	**1.25**	**39.843**	171311
		Apache	1.22	40.887	201862
		Nginx	1.22	41	**121506**
	Regular	Cherokee	6.36	7.863	158716
		Apache	**6.88**	**7.272**	216544
		Nginx	5.99	8.341	**121325**
100/10	API	Cherokee	9.76	10.246	178254
		Apache	11.47	8.715	200734
		Nginx	**11.84**	**8.444**	**121412**
	Heavy	Cherokee	1.22	82.178	191255
		Apache	2.1	47.593	215347
		Nginx	**2.25**	**44.495**	**121666**
	Regular	Cherokee	6.76	14.785	183319
		Apache	6.71	14.906	200220
		Nginx	**12.14**	**8.236**	**121402**
500/50	API	Cherokee	9.6	52.108	187319
		Apache	11.65	42.926	341887
		Nginx	**11.67**	**42.332**	**121593**
	Heavy	Cherokee	1.31	382.773	220144
		Apache	2.33	214.235	356046
		Nginx	**2.37**	**212.427**	**125128**
	Regular	Cherokee	6.65	75.147	187706
		Apache	12.35	40.5	342251
		Nginx	**12.45**	**40.173**	**121625**

Table 5.15 — continued on the next page

[16] http://www.apache.org/

Table 5.15 — continued from previous page

Req. / Conc.	Page	Proxy	Requests/s (#)	Total time (s)	Mem. Usage (B)
500/100	API	Cherokee	9.53	52.474	187887
		Apache	11.45	43.676	326127
		Nginx	**11.48**	**43.546**	**121613**
	Heavy	Cherokee	1.21	412.288	192303
		Apache	FAIL	FAIL	326197
		Nginx	**2.45**	**203.037**	**127759**
	Regular	Cherokee	6.59	75.892	191297
		Apache	12.48	40.065	326209
		Nginx	**12.54**	**39.883**	**121650**
2500/500	API	Cherokee	FAIL	FAIL	199987
		Apache	FAIL	FAIL	342211
		Nginx	FAIL	FAIL	**122012**
	Heavy	Cherokee	FAIL	FAIL	199006
		Apache	FAIL	FAIL	342472
		Nginx	FAIL	FAIL	**127524**
	Regular	Cherokee	FAIL	FAIL	195006
		Apache	FAIL	FAIL	340655
		Nginx	FAIL	FAIL	**122020**

On nonexistent concurrency, Cherokee and Apache yielded slightly better results. When concurrency is present and increases, Nginx performs slightly better. On maximum concurrency, however, none was able to cope with the demand. Regarding this last test, Cherokee failed to complete most of the requests. Nginx completed significantly more but most of them also timed out, accounting as a failed test. Apache was able to complete a few requests before becoming unresponsive, but much less than Nginx. Finally, the most impressive remark is that the memory usage of Thin with Nginx is remarkably low and impressively stable, whether it is a light API call or a heavy page. This combination brings an impressive scalability potential for its low memory usage.

5.3.2.2 Passenger Performance on Apache/Nginx

Passenger is different from the previously mentioned web servers. It is not a web server itself but instead acts as module and adds the needed functionality to Apache or Nginx to support web applications written in Ruby. Uncertainty arises when asserting whether Passenger performs better in Apache or Nginx, so a generic benchmark regarding this issue followed.

The test was similar to the one explained in the previous sections. It tested the same request/-concurrency combinations, used *ab* to measure the response rate and used the previously mentioned monitoring script to measure the memory usage. It also used 4 Passenger instances. The Passenger-specific configurations were the same on both web servers—Apache and Nginx—and can be seen on table 5.16.

Table 5.16: Passenger Options and Values

OPTION	VALUE
Spawn Method	smart
Maximum Requests	5000
Filesystem Checks Interval	5
Application Spawner Idle Time	0
Pool Idle Time	1000
Maximum Pool Size	4

The results are exhibited on table 5.17.

Table 5.17: Passenger Benchmark Results on Apache and Nginx

Req. / Conc.	Page	Web Server	Requests/s (#)	Total time (s)	Mem. Usage (B)
50/1	API	Apache	**9.37**	**5.336**	164429
		Nginx	9.1	5.496	**154437**
	Heavy	Apache	**1.18**	**42.494**	154696
		Nginx	1.16	43.072	**142983**
	Regular	Apache	6.24	8.017	185727
		Nginx	**6.36**	**7.863**	**154383**
100/10	API	Apache	12.06	8.294	207374
		Nginx	**12.28**	**8.146**	**154453**
	Heavy	Apache	**2.17**	**45.988**	168179
		Nginx	2.1	47.634	**156119**
	Regular	Apache	**12.84**	**7.79**	181126
		Nginx	11.94	8.375	**149634**
500/50	API	Apache	**11.63**	**42.979**	235130
		Nginx	11.53	43.383	**141376**
	Heavy	Apache	FAIL	FAIL	207180
		Nginx	**2.11**	**237.37**	**156234**
	Regular	Apache	**12.49**	**40.043**	200330
		Nginx	12.34	40.51	**141514**
500/100	API	Apache	FAIL	FAIL	255006
		Nginx	**11.48**	**43.546**	**141438**
	Heavy	Apache	FAIL	FAIL	220525
		Nginx	**2.1**	**238.16**	**156258**
	Regular	Apache	FAIL	FAIL	218195
		Nginx	**12.42**	**40.273**	**154676**
2500/500	API	Apache	FAIL	FAIL	238933
		Nginx	FAIL	FAIL	**154599**
	Heavy	Apache	FAIL	FAIL	220258
		Nginx	FAIL	FAIL	**143133**
	Regular	Apache	FAIL	FAIL	228932
		Nginx	FAIL	FAIL	**154699**
Total		Apache		**200.941**	199825
		Nginx		204.479	**150292**

When used with Apache, Passenger failed more tests under a rather low concurrency level—100 simultaneous requests—than when used with Nginx, which indicates that this Apache setup has some scalability issues. It should also be noticed that when paired with Nginx, Passenger uses significantly less memory.

5.3.2.3 Thin and Unicorn Performance Comparison

As mentioned before, Unicorn and Thin are very similar in their goals: they are optimized for small requests and fast clients. However, their philosophies and architecture differ. As mentioned in Section 2.3, Thin relies on *EventMachine* for fast I/O processing, using a fast asynchronous event loop. On the other hand, Unicorn forks workers which attend to one client at a time and it relies on the Operating System for many internal tasks like request queuing and worker synchronization. There are many discussions about the performance of these two web servers, becoming an issue worth addressing.

In real world applications, web servers must be ready to serve slow clients and, eventually, heavy pages. Since both web servers are not optimized for these situations, the usage of a reverse proxy is required for request and response buffering purposes.

This benchmark followed the same conditions of the aforementioned ones regarding concurrency, number of processes, test pages and the usage of *ab* to make the measurements. Thin's configuration was the same as before and Unicorn's was very similar to Thin's. Since Nginx yielded excellent results in the previous benchmarks, it was used as both web servers' reverse proxy. The results can be found on table 5.18.

Unicorn's performance is very similar to Thin's. Thin was slightly faster in total while using less memory so it yields a very slim advantaged on the performed benchmarks.

5.3.2.4 Ruby Web Servers Benchmark

The previous benchmarks addressed specific questions that the Ruby community has. The current one, however, aims at exploring all alternatives in the same environment. The results obtained for previous benchmarks narrow down possible alternatives, allowing a complete yet simple benchmark.

Nginx was used as the reverse proxy for Thin and Unicorn. It was also used as the primary web server where Passenger was docked. As seen on the previous benchmarks, it yields very similar performance to Apache and Cherokee while using less memory.

Similarly to real applications, more processes were used as well. Instead of the 4 workers used in the previous benchmarks, this test contempted 30 and 60 workers. The aforementioned developed script was used to record the memory usage in all tests. The test pages were also the same as the ones used on the previous benchmarks.

Table 5.18: Thin *versus* Unicorn Benchmark Results

Req. / Conc.	Page	Web Server	Requests/s (#)	Total time (s)	Mem. Usage (B)
50/1	API	Thin	**9.24**	**5.411**	**121213**
		Unicorn	9.17	5.45	160823
	Heavy	Thin	1.22	41	**121506**
		Unicorn	**1.25**	**39.849**	161062
	Regular	Thin	5.99	8.341	**121325**
		Unicorn	**6.89**	**7.257**	160845
100/10	API	Thin	11.84	8.444	**121412**
		Unicorn	**12.22**	**8.182**	160828
	Heavy	Thin	2.25	44.495	**121666**
		Unicorn	**2.28**	**43.832**	161035
	Regular	Thin	12.14	8.236	**121402**
		Unicorn	**12.46**	**8.023**	160865
500/50	API	Thin	**11.67**	**42.332**	**121593**
		Unicorn	11.62	43.013	160919
	Heavy	Thin	**2.37**	**212.427**	**125128**
		Unicorn	2.31	216.525	161113
	Regular	Thin	**12.45**	**40.173**	**121625**
		Unicorn	12.44	40.207	160955
500/100	API	Thin	**11.48**	**43.546**	**121613**
		Unicorn	11.46	43.618	160997
	Heavy	Thin	**2.45**	**203.037**	**127759**
		Unicorn	2.3	217.513	161207
	Regular	Thin	12.54	39.883	**121650**
		Unicorn	**12.65**	**39.516**	161027
2500/500	API	Thin	FAIL	FAIL	**122012**
		Unicorn	FAIL	FAIL	161045
	Heavy	Thin	FAIL	FAIL	**127524**
		Unicorn	FAIL	FAIL	161193
	Regular	Thin	FAIL	FAIL	**122020**
		Unicorn	FAIL	FAIL	161089
Total		Thin		**697.325**	**122324**
		Unicorn		712.985	160973

In this benchmark, however, the measurement tool was *autobench* [17], a wrapper around on HP's *httperf* [18]. Instead of continuously dumping requests to the target host like *ab* does, *autobench* runs a number of times against the specified host(s), increasing the number of requested connections *per second* on each iteration. Then it tries to find the highest request rate in which the web server remains stable and aims at maintaining that rate for the remaining simulation time, which is defined by the user.

The first round of tests was made using Ruby 1.8. The results for the *heavy page* filled with dynamic content are presented on figure 5.1.

[17] http://www.xenoclast.org/autobench/
[18] http://www.hpl.hp.com/research/linux/httperf/

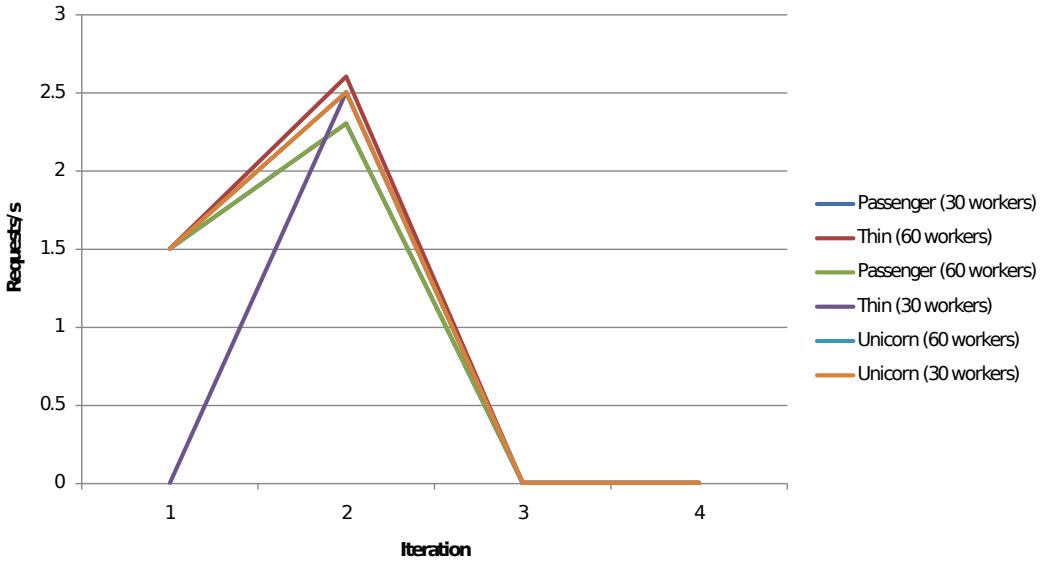

Figure 5.1: Autobench Results on the Heavy Page (Ruby 1.8)

The test tool, *autobench*, was unable to find a stable request rate on this page. All web servers behaved similarly poor.

As for the *regular page* with moderate usage of dynamic content, the results can be found on figure 5.2.

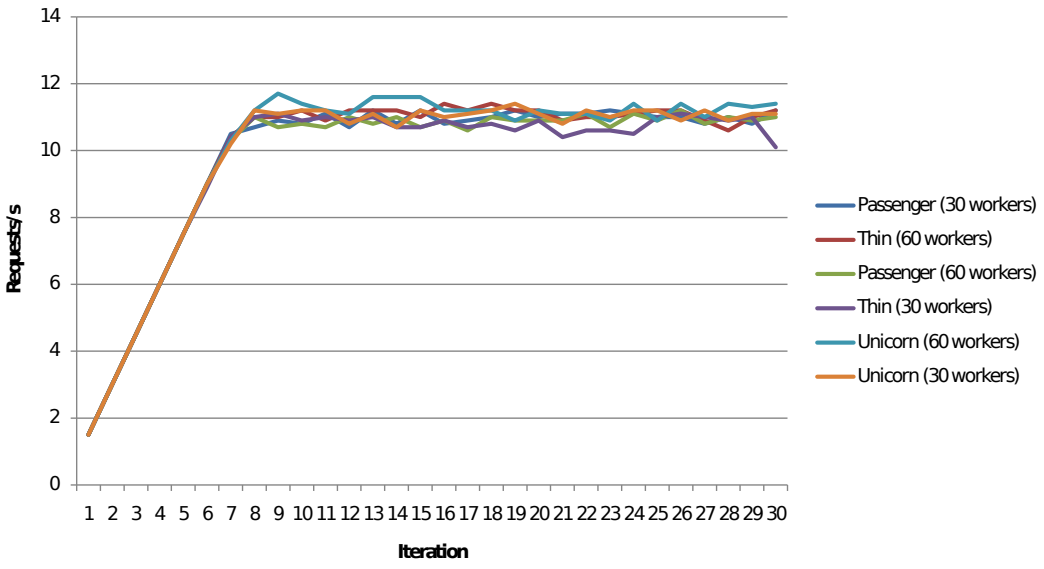

Figure 5.2: Autobench Results on the Regular Page (Ruby 1.8)

On this test all web servers were able to consistently serve requests, stabilizing at a rate of 10-12 requests *per second*. Although all had very similar performances, Unicorn with 60 workers consistently yielded slightly better results.

Finally, regarding the complex but small *API call*, the results are exhibited on figure 5.3.

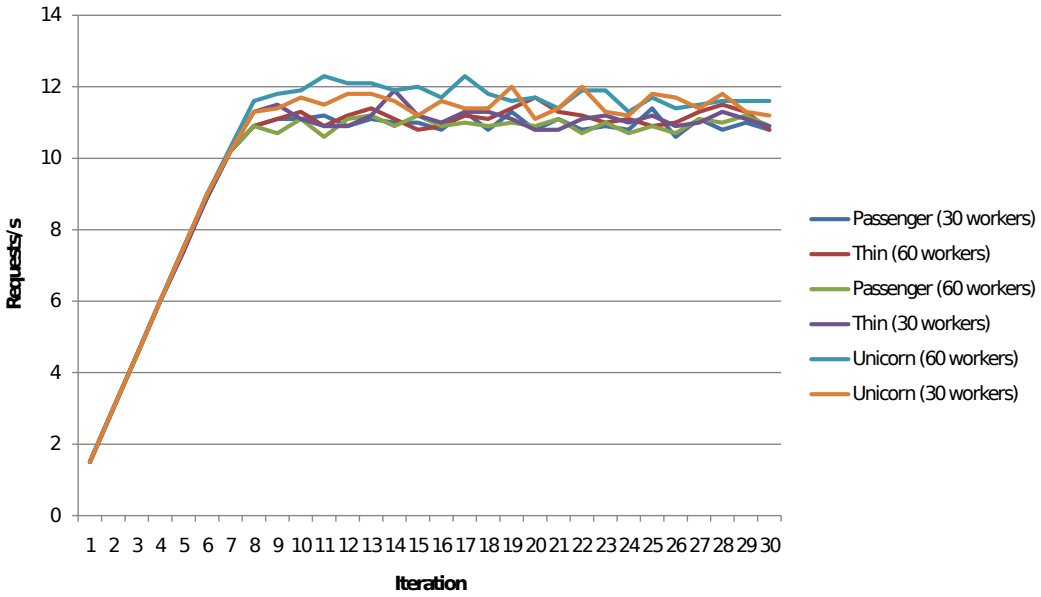

Figure 5.3: Autobench Results on the API Call (Ruby 1.8)

Once again, all web servers were able to maintain a stable request rate, which was around 10-13. Unicorn, both with 30 and 60 workers, seems to have a slight performance advantage over its competitors.

The second round was made using Ruby 1.9 on exactly the same pages. The results for the *heavy page* filled with dynamic content can be seen on figure 5.4.

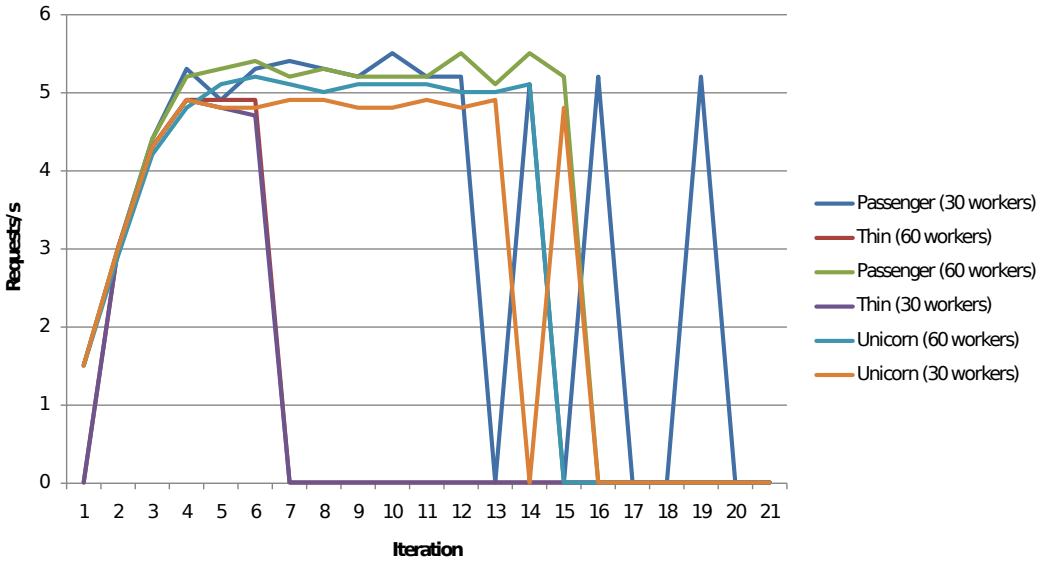

Figure 5.4: Autobench Results on the Heavy Page (Ruby 1.9)

All web servers have shown extreme improvements after switching to YARV on this page.

Most of them were stable through 15-16 iterations instead of a single one and the response rate increased from 2,5 to 4,5-5,5. Thin is the most notable exception, loosing stability after 6 iterations. However, it still represents a remarkable improvement over the previous tests with MRI.

As for the *regular page* with moderate usage of dynamic content, the results are presented on figure 5.5.

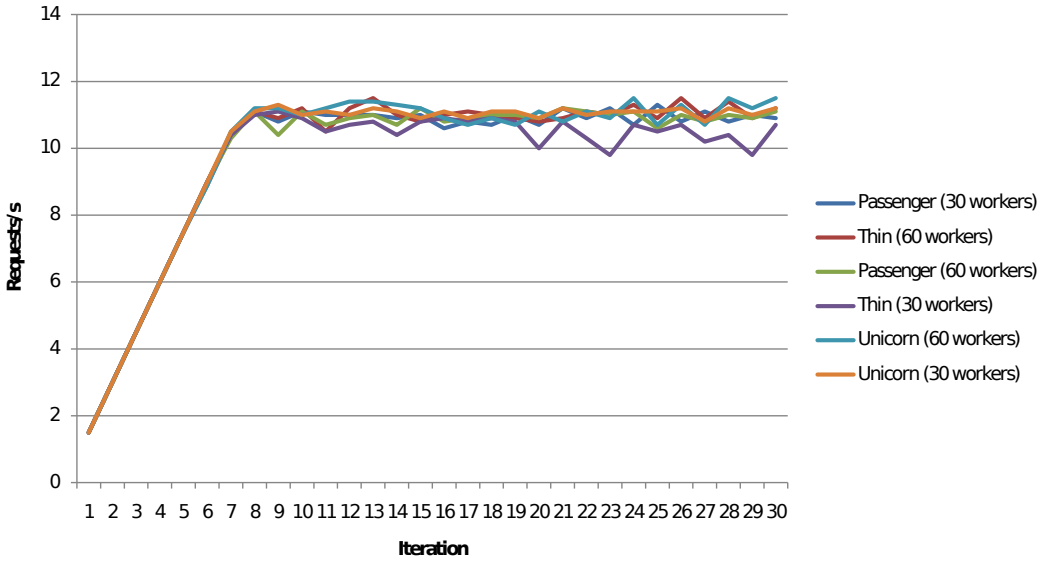

Figure 5.5: Autobench Results on the Regular Page (Ruby 1.9)

All web servers performed similarly on this test. The results were also very similar to the same test with Ruby 1.8, mainly due to the fact that this page is relatively light on Ruby code.

Finally, regarding the complex but small *API call*, the results can be found on figure 5.6.

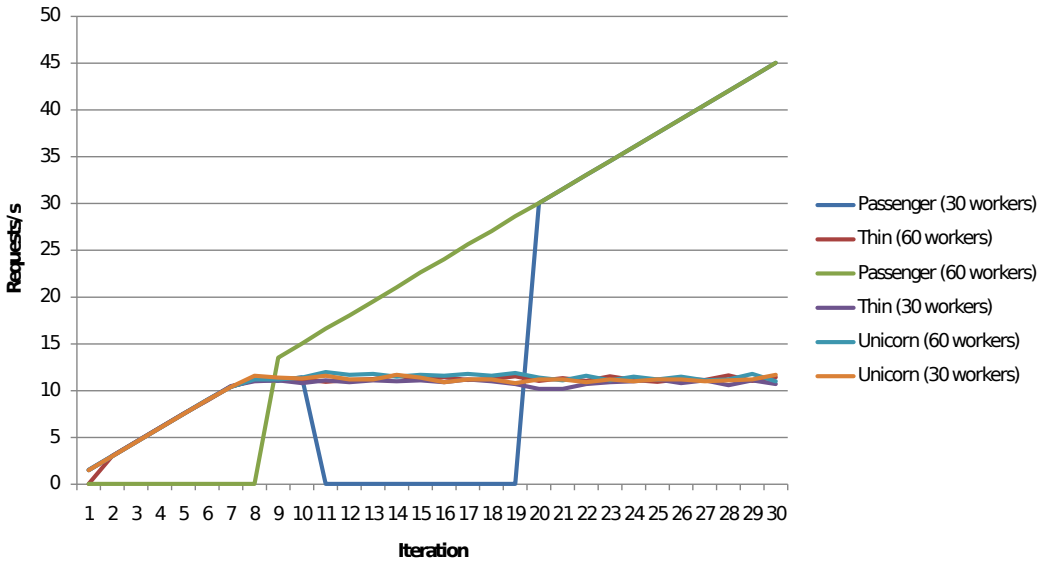

Figure 5.6: Autobench Results on the API Call (Ruby 1.9)

Once again, the results are very similar to those found on the same test with Ruby 1.8. Passenger's behavior is not consistent because its application handler segmentation faults on this specific test—complex *API call* on Ruby 1.9 with 30 and 60 workers—which is possibly related with shared resource access among workers.

Table 5.19 exhibits the average memory consumption, in megabytes, throughout all tests. A few conclusions can be drawn from the memory usage data. Firstly, Thin always uses less memory than the other web servers. Passenger's memory usage with 30 workers is similar to when it is using 60 workers, which is very high when compared to the others. Finally, using Ruby 1.9 generally causes web servers to use less memory.

Table 5.19: Web Server Memory Usage

	HEAVY PAGE		REGULAR PAGE		API CALL	
	Ruby 1.8	**Ruby 1.9**	**Ruby 1.8**	**Ruby 1.9**	**Ruby 1.8**	**Ruby 1.9**
Thin (30)	3041	2868	3018	2802	2984	2849
Unicorn (30)	3463	3556	3461	3354	3461	3442
Passenger (30)	7794	6920	7661	6650	7666	6486
Thin (60)	7214	6721	6993	6206	6995	6488
Unicorn (60)	6808	6878	6804	6566	6803	6684
Passenger (60)	7798	6921	7771	6687	7771	8045

After an exhaustive analysis of web server performance, scalability and memory usage, one can conclude that the performance differences between each alternative are very small. The area of more impact is memory usage where Thin yields the best results, closely followed by Unicorn. Using Ruby 1.9, however, induces a significant improvement in the results regarding the response rate ability and the stability of all web servers. This difference is clearly noticeable in the heaviest test of the benchmark, where most web servers have shown a $\pm 200\%$ increase in the response rate and $\pm 1500\%$ increase in successfully completed iterations.

Most of the tests have shown small to no improvements when doubling the number of workers from 30 to 60. The main bottleneck is related to the nonexistence of a memory object caching system, so increasing the workers had no effect on performance. In fact, the difference between the results of this benchmark and the previously shown ones, with only 4 workers, is remarkably small. Database caching should be in use to precisely determine the difference between having 30 and 60 workers on the aforementioned dual-core machine.

5.3.3 Tweaking

Tweaking a web server's configuration is important to fine-tune its performance. All the examined components are highly configurable and changing some settings can provoke a positive or negative impact in performance.

Apache supports two completely different MPM models: a prefork MPM, which forks a number of identical Apache processes and a worker MPM, which creates multiple threads. The prefork

model is generally more effective on systems with one or two processing units where the operating systems is better geared toward time slicing between multiple processes. When more CPUs are available, the worker model will probably be more effective. There are also important settings to address like, for instance, defining the maximum number simultaneous connections that Apache can handle using MaxClients and setting the minimum and maximum number of spare threads/processes.

Unlike Apache, Nginx does not rely on threads nor processes to handle requests. Using a philosophy very similar to Thin's, it aims at solving the C10K problem [19] by using an event-driven asynchronous architecture. Nginx supports various event models for handling connections which are optimized for certain situations and operating systems. The user must be certain that it is using the correct one for his system. One of the most efficient models—*epool*—is only supported on Linux systems running a kernel in at least version 2.6, for instance.

Similarly to Apache's worker MPM, Cherokee is a threaded web server. However, it relies on the operating system for request queuing so it also supports many polling methods which are optimized for specific systems, like *epoll*. The amount of threads can and should also be configured specifically for each application and machine.

Thin itself is not as configurable as the previously mentioned web servers. However, there are important settings like, for instance, the maximum number of connections, that can be fine-tuned. As mentioned on Section 2.3, Thin recently became capable of operating in a threaded manner by having a background pool of threads, despite its "single process single thread" philosophy. The following section will present a benchmark on this option.

Passenger has a significant amount of configuration options. One of the most important is the process spawning method. Setting it to "smart" yields the best results, according to its development team. However, it can cause incompatibilities with some *gems*. The developers must test each system and see if this spawning method is suitable or not. Other interesting options include the number of requests interval in which an application process is restarted, useful for when an application has memory leaks.

Unicorn is a very configurable web server. Its configuration is written in Ruby and there is the possibility to hook into many parts of the start and execution processes. Among all options, there is one that is particularly interesting since it can change the size of the buffers which send and receive TCP data. These can be configured to match the kernel ones—defined by *sysctl*—for optimal performance.

5.3.3.1 Thin Performance with Threading Enabled

Thin recently started to offer an option which, when activated, will cause requests to be dispatched to a background pool of 20 threads. This slightly contradicts its initial "single process single thread" philosophy, though it still relies on an asynchronous event loop in each thread.

[19] http://www.kegel.com/c10k.html

A benchmark comparing Thin's performance when threading is enabled or disabled was made using the three previously mentioned pages. Similarly to the initial benchmarks, 4 workers were used in each test and, like the initial benchmarks, *ab* as used to conduct each test. The results are exhibited on table 5.20.

Table 5.20: Thin with Threading Benchmark Results

Req. / Conc.	Page	Web Server	Requests/s (#)	Total time (s)	Mem. Usage (B)
50/1	API	Thin	**9.24**	**5.411**	**121213**
		Thin(threaded)	7.5	6.665	126491
	Heavy	Thin	**1.22**	**41**	**121506**
		Thin(threaded)	0.91	55.221	128809
	Regular	Thin	**5.99**	**8.341**	**121325**
		Thin(threaded)	4.67	10.711	126954
100/10	API	Thin	**11.84**	**8.444**	**121412**
		Thin(threaded)	11.81	8.465	128097
	Heavy	Thin	**2.25**	**44.495**	**121666**
		Thin(threaded)	1.84	54.468	133241
	Regular	Thin	**12.14**	**8.236**	**121402**
		Thin(threaded)	11.73	8.527	128709
500/50	API	Thin	**11.67**	**42.332**	**121593**
		Thin(threaded)	11.47	43.605	129938
	Heavy	Thin	**2.37**	**212.427**	**125128**
		Thin(threaded)	FAIL	FAIL	152066
	Regular	Thin	**12.45**	**40.173**	**121625**
		Thin(threaded)	11.9	42.005	130860
500/100	API	Thin	11.48	43.546	**121613**
		Thin(threaded)	**11.6**	**43.111**	130082
	Heavy	Thin	**2.45**	**203.037**	**127759**
		Thin(threaded)	FAIL	FAIL	139153
	Regular	Thin	**12.54**	**39.883**	**121650**
		Thin(threaded)	11.68	42.818	130909
2500/500	API	Thin	FAIL	FAIL	**122012**
		Thin(threaded)	FAIL	FAIL	130558
	Heavy	Thin	FAIL	FAIL	**127524**
		Thin(threaded)	FAIL	FAIL	134980
	Regular	Thin	FAIL	FAIL	**122020**
		Thin(threaded)	FAIL	FAIL	131290

Threaded Thin actually performs worse in these test pages. It uses slightly more memory and generally needs a small amount of extra time to complete each test. Threads have an associated overhead, mainly related to the spawning process and the context switches they provoke. Since Thin is based on a really fast asynchronous loop, it is likely that this processing overhead is, in this case, slowing the web server down overall.

5.3.4 Section Overview

This section exhibited and explained the work concerning Rails web servers. Regarding development, a simple yet powerful memory usage monitoring script was created. Benchmarking, however, had many phases. First of all, proxy performance involving Apache, Nginx and Cherooke was analyzed. The performance of all three was very similar, but Nginx used considerably less memory. Then, Passenger's performance on Apache and Nginx was compared. While having a similar performance across the mentioned web servers, Passenger was more scalable, more stable and used less memory under Nginx. After that, Thin and Unicorn were compared, where Thin performed slightly better while, once again, using less memory. Finally, a more complete benchmark involving Thin, Unicorn, Passenger, Nginx and different Ruby versions was made. All web servers yielded similar results, with Unicorn performing slightly better and Thin maintaining its low memory consumption. Switching from MRI to YARV, however, had a huge positive impact on all web server's performance, mainly noticed on the heaviest test. Finally, concerning tweaking, some web server options were presented and explained. Enabling threads in Thin was one of the options explored, though no performance benefits were detected.

The results obtained during all the benchmark analysis allowed to complete the general goal initially stated. Nginx poses as the best performing reverse proxy, Unicorn yields slightly better performance results under high-load and Thin consistently consumes less memory. However, these differences very scarce.

5.4 Databases

Concerning databases, the emerging MySQL Ruby library—*mysql2*—was targeted for improvements.

A database, from Rails' perspective, is all about choice. There are three different natively supported relational databases including SQLite, MySQL and PostreSQL. The community, however, as given third-party support to many alternative relational and non-relational databases including MongoDB, CouchDB, DataMapper, SQL Server and Oracle, to name a few.

The popularity of non-relational databases is increasing among the Rails' community, trading enhanced read and write speeds for higher disk usages and fewer functionalities. However, as mentioned on Section 2.4 most Rails applications rely on relational databases, mainly MySQL. On the other hand, benchmarking PostgreSQL against MySQL was discarded as a worthy task because of what was already exposed in Section 3.4. Finally, adapting *Escolinhas.pt* to a non-relational database for further evaluation would require a significant amount of refactoring and some deep architectural changes, being discarded as a possible approach to this component. For these reasons, MySQL was the chosen database to address in this thesis's context.

Due to the aforementioned Rails-centric perspective and the higher probability of success in the medium term, it was not the database itself that was targeted for improvements but instead a recent, improved Ruby database library—*mysql2*. As mentioned in Section 3.4, this Ruby MySQL

library can yield results up to 400% better than the default library in an optimal situation. However, there are caveats which need to be addressed, as explored in the following section.

Improving *mysql2* was the general goal of the work presented in this section.

5.4.1 Development

As mentioned in Chapter 4, *mysql2* handles the conversion of the data between MySQL types and Ruby objects immediately after fetching each row from the database. Since the conversion itself is done in C, there are significant performance improvements over the current default MySQL library which returns all results as "strings", forcing Rails to make the conversions in Ruby.

There are, however, caveats to the approach used by *mysql2*. Since the type conversion is not lazy casted, there is a possibility that the library is converting unnecessary data. If the developer fetches a significant amount of rows from the database but then only uses a small portion of those rows, the library is likely to have a similar or worse performance than the default library. This happens because unlike *mysql2*, using the default database driver Rails will lazy type cast the "strings" it receives, only converting the really necessary data.

Adding lazy type casting to fields in *mysql2* was developed in conjunction with one of the library's core developer, Brian Lopez. The changes are still under the core team's development process. There are many optimizations currently being made and support for some uncommon MySQL data types is still being developed and added. For this reason, the changes are not present in the current public release of this database library and there are also no reliable benchmarks to determine the real improvements over the old version of *mysql2* or other MySQL adapters. However, a patch with the changes which can be applied to the library's source code[20] is presented on appendix F.

5.4.2 Section Overview

This section exhibited and explained the work concerning databases. A promising Ruby library for MySQL was improved, by changing its behavior concerning database field casting. Instead of converting all fields from MySQL types to Ruby objects upon fetching, it now lazy type casts them as needed.

Adding lazy type casting to *mysql2* allowed to fulfill the general goal initially stated—this library's worst performing case has been reduced by solving one of its biggest caveats.

5.5 Ruby on Rails

The main focus related with Ruby on Rails itself was to improve the current tools for profiling and benchmarking applications. Motivating the adoption of the soon to be released version of this framework—Rails 3—by the community was also a goal regarding this component. Increasing

[20] http://github.com/brianmario/mysql2/

the community's awareness of this subject was also addressed by publishing an article to a popular magazine and engaging in a summer project—Ruby Summer of Code—with high visibility. Nonetheless, some common pitfalls were also addressed while presenting and benchmarking their solutions.

Improving Rails' profiling tools, motivating the adoption of Rails 3 and increasing the community's awareness of this subject were the general goals of the work presented in this section.

5.5.1 Benchmarking

Rails developers can often incur in some development performance pitfalls which can easily be addressed by using some of Rails' powerful functionalities. This section will cover some of these possible pitfalls, propose solutions and generally assert their performance differences.

5.5.1.1 Eager Loading

Eager loading is a core feature of Ruby on Rails which, in certain situations, can lead to significant performance slowdowns because of its absence or misuse. It allows the developer to specify if and which associations should be loaded up front, being the opposite of lazy loading. A classical example of the usefulness of this feature consists in a blog post with comments in which rendering the post's page will also render its comments. Eager loading allows the developer to specify that the comments should be loaded alongside the post, instructing Rails to only use two database queries—one for the post and another for its comments. If not explicitly specified, Rails will load each comment independently while rendering them. Loading the post and eager loading its comments is illustrated in the following code.

```
1 @posts = Post.find(:all, :include => :comments)
```

A simple benchmark was performed on a scaffold blog Rails application. The Rails version in use was 2.3.5. The database in use was MySQL using Ruby's default library. A thousand blog posts were generated automatically, along with two hundred thousand comments randomly associated with blog posts. This benchmark compares the time needed to complete a request which fetches all posts along with their comments. The test was run five times to eliminate circumstantial issues and the best results are shown. The results are exhibited on table 5.21.

Table 5.21: Eager Loading Benchmark Results

EAGER LOADING	TIME (SECONDS)
No	283
Yes	21

In this extreme example, eager loading the data took $\pm 7\%$ of the time needed to lazy load the data, which is Rails default behavior. Eager loading can have a significant performance impact, depending on the context, as exemplified.

5.5.1.2 Transactions

Rails wraps every database write and update inside a transaction if not instructed otherwise. This happens because of its before/after filter functionality, which allows developers to hook code before and after certain actions are executed. In these cases, the operation and its filters are wrapped inside a transaction.

There are cases, however, where this behavior is suboptimal. When write/update calls are being executed consecutively—for instance, inside a loop—all database calls will be inside their own transaction. This will incur in significant performance penalties, depending on the amount of database actions. However, Rails allows developers to specify where the transaction should begin and end. The following code exemplifies this statement.

```
1 ActiveRecord::Base.transaction do
2   (1..100).each { |i| Number.create(:value => i) }
3 end
```

In this example, instead of using one hundred transactions Rails will only use a single one. Using the same environment from the benchmark in the previous section, a benchmark was conducted on asserting the performance difference in explicitly using a global transaction instead of letting Rails place many small transactions which is, as previously mentioned, its behavior by default. This benchmark consisted in creating posts and associated comments, all with the same content. Two tests were made, the first by creating one hundred posts with twenty comments each and a second one by creating one post with two comments. Results can be seen on table 5.22.

Table 5.22: Explicit Transaction Benchmark Results

	EXPLICIT TRANSACTION	TIME (MILLISECONDS)
100 posts, 20 comments	No	76017
100 posts, 20 comments	Yes	5223
1 post, 2 comments	No	176
1 post, 2 comments	Yes	137

Using a global transaction on the heaviest test implied a $\pm 93\%$ reduction on the time needed to write the data. On the lighter test, however, this reduction was of $\pm 22\%$. On consecutive writes and/or updates, using a global transaction can significantly increase the performance of the operation.

5.5.1.3 Magic Finders

Rails has a feature commonly called “magic finders”. It improves code’s readability by allowing the developer to replace some calls with smaller and more readable ones. The following example illustrates this feature, comparing with a regular call.

```
1 Comment.find(:first, :conditions => ["created_at = ?", "2009-11-06
    18:25:48"]) # normal find
2
3 Comment.find_by_created_at("2009-11-06 18:25:48") # magic find
```

These calls, however, have an associated performance penalty. Using the same test environment found on the previous benchmarks, both previously presented calls were benchmarked. The results are presented on table 5.23.

Table 5.23: Magic Finders Benchmark Results

TYPE	TIME (MILLISECONDS)
Normal find	62
Magic find	197

The normal find only takes ±32% of the time needed by the magic find to complete. Since there is a readability/performance trade-off, using either syntaxes is a decision which depends on the project and its developers. The general performance penalty associated with the commonly used magic finders is, however, significant and should be taken into account.

5.5.1.4 Fetching Large Groups of Records

In certain situations, Rails applications need to fetch a significant amount of rows from the database, instantiate each model object and render them in a view. Using the "find" helper, this operation can be significantly heavy on memory, as it fetches and loads all records into Ruby objects, and will lock the application until all operations are complete.

However, when Rails 2.3 was introduced it bundled a couple of methods which are likely to be very useful in these situations: "find_each" and "find_in_batches". The first will retrieve a specified amount of objects at a time, letting the code iterate over the records as if it was a normal "find" call. The second is very similar, except that it gives control back to the application every time it fetches a batch. Both methods have a "batch_size" option which defaults to one thousand records. These additional methods have the advantage of using less memory and, as explored in Section 3.2, less memory usage will trigger Ruby's garbage collector less often, providing better execution times. Notwithstanding these advantages, these methods are unlikely to yield performance improvements when fetching small amounts of records.

To accurately determine the performance benefits involved with switching the "find" call with "find_each" when there is a significant amount of records involved, a benchmark was done. The environment was the same as previous benchmarks. The test itself consisted in fetching the two hundred thousand comments in the example blog application. The results can be found on table 5.24.

Using the alternative method to fetch records in batches, the application needed 10% less time to fetch the records and used 25% less memory. These are significant improvements, mainly in memory usage.

Table 5.24: Fetching Records in Batches Benchmark Results

METHOD	TIME (SECONDS)	MEMORY USAGE (MB)
find	211	158
find_each	195	118

5.5.2 Development

The development phase regarding this component involved many distinct activities. First of all, it was crucial to refactor Rails' native profiling and benchmarking tools. Taking into consideration that the release of the new version of the Ruby on Rails framework is nearing, it was important to motivate its early adoption by porting some widely used plugins/gems and one of this framework's most famous applications to the most recent version. After that, a small task related to adding support for the Nginx's *X-Accel-Redirect* header to Rails was performed. Finally, an overview on the current work under the Ruby Summer of Code program is given.

5.5.2.1 Refactoring Rails Profiling/Benchmarking Tools

Rails' profiling and benchmarking tools were nonfunctional on YARV, as they relied on Rub 1.8's now outdated architecture. Ruby evolved very fast and these tools did not keep up, becoming deprecated and useless when used on this version.

Adding Ruby 1.9 support to these tools was crucial, since developers could not rely on the native tools when benchmarking and profiling their applications. By taking advantage of the enhances made to Ruby's profiling tools mentioned in Section 5.2, Rails was overhauled [21] and now fully supports profiling and benchmarking under Ruby 1.9.

5.5.2.2 Porting Plugins to Rails 3

Rails 3 adoption by developers is highly conditioned by plugin availability. As mentioned in Section 2.5, there is a significant amount of plugins available for Ruby on Rails. However, due to the recent Rails' architectural changes, authors generally need to update their plugins/gems to be compatible to this version. The lack of available plugins will certainly limit the adoption rate of this version. In order to help alleviating this problem, some famous plugins were updated to comply with the new Rails API for plugins/gems. All plugins used by *Escolinhas.pt* were analyzed and the ones with the higher watcher count[22] on their public repositories, were selected. Their names and functionality are:

acts_as_list, a plugin for sorting and reordering a number of objects in a list[23];

permalink_fu, a plugin for creating URL-friendly permanent links from object attributes[24];

[21] http://github.com/rails/rails/commits/master?author=goncalossilva
[22] GitHub allows users to start "watching" repositories, being notified of all changes made to the projects they "watch"
[23] http://github.com/goncalossilva/acts_as_list/
[24] http://github.com/goncalossilva/permalink_fu/

acts_as_paranoid, a plugin for hiding records instead of deleting them, being able to recover them[25].

All the mentioned plugins were refactored to use the new plugin API. Some tests also needed refactoring, as they used deprecated Rails code. However, updating *permalink_fu* also involved replacing its character conversion engine since it relied on deprecated Ruby 1.8 functionality, meaning this plugin was not compatible with Ruby's most recent version. On the other hand, *acts_as_paranoid* had to be completely rewritten. It was initially developed for Rails 1 and slightly patched to work on Rails 2, having a deprecated architecture which was incompatible with Rails 3. Recreating this plugin from ground up with Rails 3 in mind allowed a $\pm$70% reduction in lines of code.

5.5.2.3 Porting Redmine to Rails 3

Redmine[26] is an open-source flexible project management web application written in Ruby on Rails, created by Jean-Philippe Lang. It is one of the most famous open-source Rails projects, having more than 90 reported major installations worldwide [Lan], including the development teams of Ruby, phpBB and Lighttpd. It is a complex platform with over 60 models and 65000 lines of code.

Redmine was upgraded to be compatible with Rails 3 and the source code is hosted at GitHub[27]. This upgrade increased Rails 3's visibility while showing the potential performance improvements in upgrading, since this version is reportedly faster. The benefits can be experienced by the users themselves while using Redmine. Many issues had to be addressed, including fixes on the application initialization process, string handling, parameter filtering, plugin support, usage of Javascript helpers and method overriding, among others. All changes can be explored in detail on the project's commit log at GitHub.

5.5.2.4 Adding X-Accel-Redirect Support to Rails

Rails lacks support for Nginx's *X-Accel-Redirect* header by default, so a plugin to add it was developed[28]. Similar to Apache's *X-Sendfile*, *X-Accel-Redirect* can be useful when a user requests the download of a file from the server. If this flag is set on the response body, it will instruct Nginx to handle the file delivery itself. This way, the Rails process will be free to handle other requests instead of blocking to serve the file, delegating that task to Nginx.

[25] http://github.com/goncalossilva/rails3_acts_as_paranoid/
[26] http://www.redmine.org/
[27] http://github.com/goncalossilva/redmine
[28] http://github.com/goncalossilva/X-Accel-Redirect

5.5.2.5 Benchmarking Continuous Integration

Late in this project's course, applications for the Ruby Summer of Code[29] began. This program is similar to Google Summer of Code, being a student internship program designed to help fund student development of Ruby-related projects during the summer.

The decision to apply for this program was made, by proposing the development of a Benchmarking Continuous Integration for Ruby on Rails. This proposal was later accepted and a project mentor assigned—Yehuda Katz. Yehuda, the lead developer of the discontinued Merb framework, is one of the most prominent developers behind Rails 3, having joined the core team as soon as the decision to merge Rails and Merb became official.

"Project #8: Benchmarking CI[30]" consists in building an official full-stack benchmarking suite for Ruby on Rails. This way, each commit to the Rails repository will automatically trigger a process where a remote machine starts a new server, runs the tests and reports back the results. As time goes by, it will be possible to watch the evolution of the framework's performance and developers will be able to keep track of the impact their changes have. Any performance regressions will be detected and the responsible developers notified. In its most basic form, it will bring a kind of performance-oriented continuous integration for the Ruby on Rails framework.

The whole Rails community will benefit from having an official benchmarking suite for Ruby on Rails. On one hand, developers will have increased awareness of the framework's performance. They will be able to benchmark their changes and understand their impact on every component, making small adjustments if any significant performance regressions are found. On the other hand, the community will also benefit since Rails will definitely become faster and more scalable over time.

5.5.3 Community Blog

Soon after the start of the project, a public blog named "Snap Rails"[31] was created. Many of the findings done throughout this project have been presented there. Despite being fairly recent, it already surpasses a thousand unique visitors. A map overlay showing their locations is shown on figure 5.7. Visitor analysis was made using Google Analytics[32]. With visitors from more than fifty five countries, this blog's purpose is to continuously increase this subject's visibility.

5.5.4 Performance-oriented Article Series

A series of performance-oriented articles for the *Rails Magazine* has also begun. The first article is already present on the sixth edition of the magazine [Sil10], with more to follow on future releases. Similarly to this thesis, this article performance series will cover all system's components, having already started with an introduction to this subject. A copy is exhibited in appendix G.

[29] http://www.rubysoc.org/
[30] http://www.rubysoc.org/projects/
[31] http://snaprails.tumblr.com
[32] http://analytics.google.com

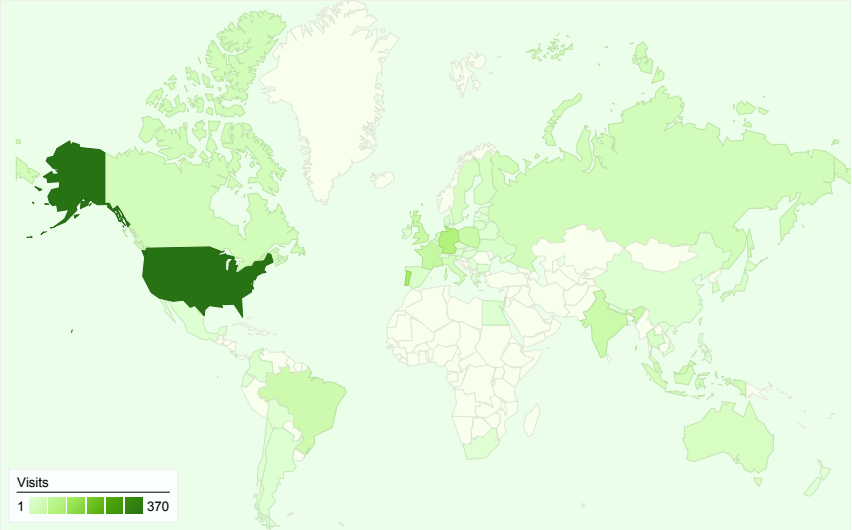

Figure 5.7: “Snap Rails” Map Overlay

Publishing articles on this matter on such a popular magazine increases this subjects’ visibility and the importance in building highly scalable Ruby on Rails applications, possibly igniting the community’s awareness of this subject.

5.5.5 Section Overview

This section exhibited and explained the work concerning Ruby on Rails. Regarding benchmarking, common performance pitfalls and their solutions were analyzed. Concerning development, Rails’ profiling and benchmarking tools were revamped and now seamlessly integrate with Ruby’s. After that, some renowned plugins and Redmine were ported to Rails 3. A plugin which adds support for Nginx’s *X-Accel-Redirect* was also created. Finally, a project related to the creation of a performance-oriented continuous integration for Rails under the Ruby Summer of Code program was detailed. This section also presents some work that does not fit under the usual fields—benchmarking, development and tweaking—which is related with increasing the community’s awareness of the importance of building highly performant Rails applications. This work is related with the creation of a community blog on this subject and the start of a performance-oriented series of articles for *Rails Magazine*.

Making Rails’ profiling tools agnostic to which Ruby version is using them and consequently seamless integrate with YARV allowed to improve these tools. Porting Redmine and various plugins to Rails 3 increased this version’s visibility and eased the process of upgrading, motivating its adoption. Creating the general guidelines by using information from this and previous sections, starting a community blog and writing an article series are likely to increase the community’s awareness of this subject. Finally, building a benchmarking continuous integration for Rails itself will also increase its developer’s awareness of this subject, completing the last of the general goals initially stated.

Chapter 6

Conclusions

This section presents the conclusions gathered from this project, a summary of contributions and what may be done to extend and improve this work in the future.

6.1 Results

The main objectives described in Section 4 have been fulfilled.

General guidelines and conventions for building highly performant and scalable Ruby on Rails applications have been created. The research performed to elaborate the technologies overview and the state of the art, exhibited in chapters 2 and 3, supplied important knowledge that shaped the initial decisions on what to further research. The conclusions drawn from the benchmarking phases of each component's analysis and their application to *Escolinhas.pt* allowed to fill in some the flaws found in the initial research, providing a solid knowledge base for the elaboration of the aforementioned guidelines and conventions. They can be found in appendix H and an online version[1] is also available.

The native profiling tools for Ruby and Rails applications have been revamped and brought up to date. Rails' profiling tools were refactored in order to enable support for YARV, the official interpreter for Ruby 1.9. On the other hand, improvements were also made on YARV concerning this subject, by recording more information and enhancing its information retrieval and presentation capabilities.

The global awareness of the importance in building highly performant and scalable Ruby on Rails applications also increased. Creating a general guide, revamping the native profiling tools, adding configuration options to YARV's garbage collector, starting a public blog on this subject, writing a series of articles oriented towards performance for Rails Magazine and developing an official benchmarking continuous integration suite for Rails, all contribute to an increased awareness of this matter, from the core team of Rails to the web developers themselves. Given the

[1] http://goncalossilva.com/scaling_rails.html

performance advantages of the latest versions of Rails, porting Redmine to Rails 3 increases the visibility of these benefits since Redmine's users are able to experience it themselves. Finally, adding support for Ruby 1.9 and Rails 3 to some famous plugins also lowers the porting effort for some applications.

All these contributions positively affect Rails' performance and scalability. Some activities had direct impact on the performance of Rails-related components, like improving adding adaptive performance to YARV or adding lazy type casting to *mysql2*. Others had indirect impact, like enhancing profiling tools or developing a benchmarking suite for Rails. Some of the remaining activities focused on evaluating the performance of component alternatives, exploring their configurations and making the results and conclusions public. All this work enabled a final but very important conclusion: general centralized information is more accessible, profiling Rails' applications became easier and the global awareness of the importance of this subject also increased.

6.2 Summary of Contributions

The research and work performed in this thesis' context allowed to create new tools and contribute with new knowledge to the Ruby on Rails community.

First of all, generic guidelines and conventions on building highly performant and scalable Ruby on Rails applications were created. They are exhibited on appendix H, also being available on the internet[2].

Rails' native profiling and benchmarking tools got overhauled and now fully support Ruby 1.9.

Ruby 1.9's internal profiler was improved. It now stores more information than previously and is able to yield internal data in an appropriate format for automated processing, the *hash* format, while still supporting the *string* format. On another matter, its garbage collector is now flexible, offering five configuration parameters for adaptive performance.

Stephen Kaes' HTML printer for profiling Ruby applications was refactored to be compatible with recent Ruby interpreters, from 1.8.7 to 1.9.2. It was also integrated with the "ruby-prof" code profiler.

Escolinhas.pt—the main test application—is now running on Ruby 1.9, notoriously benefiting from this version's performance improvements.

Redmine—the aforementioned open-source project management tool—is now compatible with Rails 3, increasing the visibility of this version's benefits as a wide range of users are using this tool and are now able to personally experience the differences.

Some reknowned plugins—acts_as_list, acts_as_paranoid and permalink_fu—were refactored and now use the new Rails API thus being compatible with Rails 3. On a related matter, permalink_fu no longer requires Ruby 1.8 to function properly, as its character conversion engine has been completely rewritten and is now compatible with Ruby 1.9.

[2] http://goncalossilva.com/scaling_rails.html

The first article of a performance-oriented series called “Scaling Rails” was published in the *Rails Magazine*.

Support for lazy type casting was added to *mysql2*, an emerging MySQL Ruby library.

A community blog tackling Rails’ performance and scalability was started.

A simple and lightweight utility for measuring the memory usage of multiple processes on UNIX systems was developed. A generic benchmarking script oriented towards operating system performance has also been created. It uses third-party tools which are cross-platform compatible except for “hdparm”, which is disabled if not running on a Linux system.

The performance of some server-oriented Linux distributions—Debian, Ubuntu Server, CentOS and Gentoo—while performing general tasks was analyzed, using the aforementioned benchmarking script. The performance of Ruby 1.8 and 1.9 was compared and demonstrated on multiple operating systems as well. Possible configuration options of Linux systems were also explored.

The performance of multiple Ruby web servers was exhaustively compared, including an analysis of some distinct configuration options found in each web server and their impact on its behavior.

Finally, the performance impact of some Ruby on Rails features was explored, namely: eager loading, transactions, magic finders and fetching large groups of records.

6.3 Ongoing Work

Some of the activities started during the course of the project are not yet fully complete. This section explores them, giving an overview of the remaining work.

6.3.1 Lazy Type Casting in mysql2

This Ruby library now lazy type casts database values. However, support for less common MySQL data types is currently being enhanced. This optimized version of *mysql2* will be released as soon as the testing phase ends.

6.3.2 Community Blog

The aforementioned community blog is an ongoing project, as more content is being frequently added and future work will be publicly presented there.

6.3.3 Performance-oriented Article Series

Notwithstanding the already published article, there are others currently being written and improved for further releases of *Rails Magazine*.

6.3.4 Benchmarking Continuous Integration

The development of the official performance-oriented continuous integration process for Ruby on Rails started at the end of May and will continue until mid-August. This is the official project schedule of Ruby Summer of Code, the program under which this project is being developed.

6.4 Future Research

Many ideas were considered during the course of the project, but some of them could not have been pursued within the available time frame. This section presents some of these ideas, which can be pursued by the thesis' author or other researchers in the future.

6.4.1 Approaching Non-Relational Databases

The current research has focused on the "Ruby component" of the most popular database among Rails' developers—Ruby's *MySQL* library. However, it would be interesting to explore the benefits of using a non-relational database, especially when taking into account the feedback some people are giving on the performance improvements these databases provide. Since these require the usage of specific ORM's, adding native support in Active Record for some of these databases—namely MongoDB and CouchDB—would probably increase their adoption among the developers.

6.4.2 Generational Garbage Collection

As previously mentioned, one of the main issues in Ruby, from the perspective of a Ruby on Rails application, is its garbage collector. It is based on the simple *mark-and-sweep* algorithm which is not very efficient for Rails applications, which frequently allocate and free considerable amounts of memory. Other interpreted languages such as Java and Python have a generational garbage collector, known for its superior efficiency. It would require a significant amount of effort to implement such algorithm on the Ruby interpreter since some C extensions rely on the current GC's behavior. However, it would be very interesting to analyze a working implementation of a generational garbage collector for Ruby from a performance standpoint.

6.4.3 Native Caching

Caching is a recurring topic when optimizing web applications. Knowing that the database is often one the major bottlenecks found on these applications and considering that caching significantly decreases this bottleneck's impact, it would be interesting to add native caching support in Rails. While Rails has support for some reknowed tools like *Memcached*[3], it natively lacks this support. Using third-party tools generally implies an extra effort since they are not as seamlessly integrated with Rails as a core component. *Memcached*, for instance, requires that the developers use its methods to fetch the data from cache and, on the other hand, use Rails' methods to fetch data from

[3] http://memcached.org/

the database. It also requires that the developers explicitly store the data in the cache every single time it changes and none of these procedures is automatic. However, if it was natively integrated, it would be possible to develop an optimized solution in which the developer would simply fetch the data, not needing to know if it was already cached, expired or any other specific detail provoked from using two separate interfaces for fetching data, depending on its location. This way, Rails would handle the caching and retrieval processes automatically.

6.4.4 Alternative Ruby Interpreters

Many alternative Ruby interpreters are nearing 100% compatibility with the Ruby 1.9 specification, namely *JRuby* and *Rubinius*. In a near future, these will be available as full-fledged Ruby implementations worth benchmarking and comparing with YARV, the official interpreter for the 1.9 specification.

6.4.5 Rewriting Web Servers in C

Some of the most successful Ruby web servers—namely Thin and Unicorn—are mostly written in Ruby. As previously mentioned, Mongrel's performance is significantly superior to WEBrick's and the only difference is its reimplementation of the HTML parser in C. There is a high possibility that implementing some core parts of Thin and Unicorn in C would also result in a significantly improved performance.

6.5 Global Impact

The course of "Scaling Rails: a system-wide approach to performance optimization" aided establishing a performance-oriented mindset within the Rails community. All system components were analyzed and improved. The findings were and will continue to be shared via blog posts and magazine articles. Improved solutions to given bottlenecks, features and components were and will continue to be published. The Rails community has a strong foundation in open-source software development, allowing all this to happen at a very fast rate. This subject's popularity greatly increased, something highly noticeable within the Rails core development team with whom I am gearing towards fine-tunning Rails' performance even further. Components were enhanced, information was gathered and made publicly accessible. The state of the art in Rails scalability is now improved. Ruby on Rails applications can now easily increase their performance and scalability.

CONCLUSIONS

References

[10ga] 10gen. Comparing mongodb and couchdb. http://www.mongodb.org/display/DOCS/Home. Accessed 15th June 2010.

[10gb] 10gen. Home - mongodb. http://www.mongodb.org/display/DOCS/Home. Accessed 15th June 2010.

[37s] 37signals. Basecamp turns 1,000,000 - (37signals). http://37signals.com/svn/posts/106-basecamp-turns-1000000. Accessed 15th June 2010.

[AG] Paessler AG. Comparing php script performance on linux and windows. http://www.paessler.com/webstress/sample_performance_tests/comparing_php_script_performance_on_linux_and_windows. Accessed 15th June 2010.

[AG07] Donovan Artz and Yolanda Gil. A survey of trust in computer science and the semantic web. *Web Semant.*, 5(2):58–71, 2007.

[AI] Inc. Alexa Internet. Alexa the web information company. http://www.alexa.com/. Accessed 15th June 2010.

[Ali09] Muhammed Ali. Ruby 1.9.x web servers booklet: A survey of architectural choices and analisys of relative performance. Octorber 2009.

[Apa] Apache Software Foundation. Apache core features. http://httpd.apache.org/docs/2.2/mod/core.html. Accessed 15th June 2010.

[App] Inc. Apple. Leopard technology series for developers: Os foundations. http://developer.apple.com/leopard/overview/osfoundations.html. Accessed 15th June 2010.

[Auz09] Jun Auza. 5 best bsd distributions. http://www.junauza.com/2009/04/5-best-bsd-distributions.html, March 2009. Accessed 15th June 2010.

[AWS] LLC. Awio Web Services. W3counter - global web stats. http://www.w3counter.com/globalstats.php. Accessed 15th June 2010.

[Bab06] Charles Babcock. What's the greatest software ever written? http://www.informationweek.com/shared/printableArticle.jhtml;jsessionid=VTSZGLDLQNJ55QE1GHOSKH4ATMY32JVN?articleID=191901844, August 2006. Accessed 15th June 2010.

[BK07] Michael Bächle and Paul Kirchberg. Ruby on rails. *IEEE Softw.*, 24(6):105–108, 2007.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[BS10] John Adams Ben Sandofsky. Unicorn power. http://engineering.twitter.com/2010/03/unicorn-power.html, March 2010. Accessed 15th June 2010.

[Can08a] Antonio Cangiano. Shotgun rewrite underway. http://betterruby.wordpress.com/2008/04/11/shotgun-rewrite-underway/, April 2008. Accessed 15th June 2010.

[Can08b] Antonio Cangiano. Shotgun the rubinius virtual machine. http://betterruby.wordpress.com/2008/03/18/shotgun-the-rubinius-virtual-machine/, March 2008. Accessed 15th June 2010.

[Can09a] Antonio Cangiano. Comparing the performance of ironruby, ruby 1.8 and ruby 1.9 on windows. http://antoniocangiano.com/2009/08/03/performance-of-ironruby-ruby-on-windows/, August 2009. Accessed 15th June 2010.

[Can09b] Antonio Cangiano. Holy shmoly, ruby 1.9 smokes python away! http://antoniocangiano.com/2007/11/28/holy-shmoly-ruby-19-smokes-python-away/, November 2009. Accessed 15th June 2010.

[Can09c] Antonio Cangiano. How much faster is ruby on linux? http://antoniocangiano.com/2009/08/10/how-much-faster-is-ruby-on-linux/, August 2009. Accessed 15th June 2010.

[Can09d] Antonio Cangiano. Shotgun rewrite underway. http://antoniocangiano.com/2009/03/23/rubys-biggest-challenge-for-2009/, March 2009. Accessed 15th June 2010.

[Cec09] Emmanuel Cecchet. Building petabyte warehouses with unmodified postgresql, May 2009. Keynote Presentation.

[cha] Changes in Ruby 1.9. http://eigenclass.org/hiki.rb?Changes+in+Ruby+1.9. Accessed 15th June 2010.

[Cha09] Scott Chacon. *Pro Git*. Apress, Berkely, CA, USA, 2009.

[com09] Inc. comScore. Hulu continues ascent in u.s. online video market, breaking into top 3 properties by videos viewed for first time in march. http://www.comscore.com/Press_Events/Press_Releases/2009/4/Hulu_Breaks_Into_Top_3_Video_Properties, April 2009. Accessed 15th June 2010.

[Cora] Oracle Corporation. Differences between mri and jruby. http://kenai.com/projects/jruby/pages/DifferencesBetweenMriAndJruby. Accessed 15th June 2010.

[Corb] Oracle Corporation. Mysql 5.5 reference manual :: 2.3.15 mysql on windows compared to mysql on unix. http://dev.mysql.com/doc/refman/5.5/en/windows-vs-unix.html. Accessed 15th June 2010.

[Corc] Oracle Corporation. Why mysql? http://www.mysql.com/why-mysql/. Accessed 15th June 2010.

[Cou] Marc-André Cournoyer. Thin - yet another web server. http://code.macournoyer.com/thin/. Accessed 15th June 2010.

[Cri09] James Crisp. Rails 3: Vaporware to awesome. http://jamescrisp.org/2009/02/21/linux-virtualbox-vs-windows-for-rails-dev/, February 2009. Accessed 15th June 2010.

[Dym09] Alexander Dymo. The future of rails is ruby 1.9 - real performance of 1.8, jruby and 1.9 compared. http://blog.pluron.com/2009/05/ruby-19-performance.html, May 2009. Accessed 15th June 2010.

[DZ65] J. Dobbie and D. Zatyko. System optimization: A mass memory system designed for the multi-program/multi-processors users. In *Proceedings of the 1965 20th national conference*, pages 487–500, New York, NY, USA, 1965. ACM. Chairman-Arden, B. W.

[eMa] Inc. eMarketer. Us twitter usage surpasses earlier estimates - emarketer. http://www.emarketer.com/Article.aspx?R=1007271. Accessed 15th June 2010.

[eve01] everything2.com. Bsd code in windows. http://www.everything2.com/index.pl?node=BSD%20Code%20in%20Windows, March 2001. Accessed 15th June 2010.

[EY] Inc. Engine Yard. Rubinius: Use ruby. http://rubini.us/. Accessed 15th June 2010.

[EY09] Inc. Engine Yard. 5 tips to scale your ruby on rails application. http://www.engineyard.com/blog/2009/5-tips-to-scale-your-ror-application/, May 2009. Accessed 15th June 2010.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The ruby intermediate language. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 89–98, New York, NY, USA, 2009. ACM.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008.

[Got05] G. Goth. Open source business models: ready for prime time. *Software, IEEE*, 22(6):98–100, Nov.-Dec. 2005.

[Gri06] Lenz Grimmer. Interview with david heinemeier hansson from ruby on rails. http://dev.mysql.com/tech-resources/interviews/david-heinemeier-hansson-rails.html, Feburary 2006. Accessed 15th June 2010.

REFERENCES

[Gri08] Ilya Grigorik. 6 optimization tips for ruby mri. http://www.igvita.com/2008/07/08/6-optimization-tips-for-ruby-mri/, July 2008. Accessed 15th June 2010.

[Gro] Miniwatts Marketing Group. World internet usage statistics news and world population stats. http://www.internetworldstats.com/stats.htm. Accessed 15th June 2010.

[Hana] David Heinemeier Hansson. Ruby on rails. http://rubyonrails.org/. Accessed 15th June 2010.

[Hanb] David Heinemeier Hansson. Ruby on rails: Applications. http://rubyonrails.org/applications. Accessed 15th June 2010.

[Han10] David Heinemeier Hansson. rails/activesupport/lib/active_support/testing/performance.rb. http://github.com/rails/rails/blob/606203c03494168c03fa81b5fdf71ee8ede7faf0/activesupport/lib/active_support/testing/performance.rb, January 2010. Accessed 15th June 2010.

[HF86] Robert Brian Hagmann and Domenico Ferrari. Performance analysis of several back-end database architectures. *ACM Trans. Database Syst.*, 11(1):1–26, 1986.

[HH07] Curt Hibbs and Brian Hogan. *Rails on windows*. O'Reilly, 2007.

[Hof06] Jason Hoffman. Scaling a rails application from the bottom up. Rails Conf Europe 2007, September 2006. Keynote Presentation.

[Ice08] Boxed Ice. Garbage collection is why ruby on rails is slow: Patches to improve performance 5x; memory profiling. http://blog.pluron.com/2008/01/ruby-on-rails-i.html, January 2008. Accessed 15th June 2010.

[Ice09] Boxed Ice. Choosing a non relational database: why we migrated from mysql to mongodb. http://goo.gl/O9aA, July 2009. Accessed 15th June 2010.

[IFVDCI08] José Ignacio Fernández-Villamor, Laura Díaz-Casillas, and Carlos Á. Iglesias. A comparison model for agile web frameworks. In *EATIS '08: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[Igo] Igor Sysoev. nginx. http://nginx.org/en/. Accessed 15th June 2010.

[Inf09] Inc. InfoEther. Ruby and rails ecosystem white paper. Technical report, Herndon, Virginia, USA, 2009.

[int] The Ruby VM interview. The Ruby VM: Episode III. http://blog.grayproductions.net/articles/the_ruby_vm_episode_iii. Accessed 15th June 2010.

[IT 09] IT Facts. 58% of americans have a mobile phone with web connectivity. http://www.itfacts.biz/58-of-americans-have-a-mobile-phone-with-web-connectivity/12965, May 2009. Accessed 15th June 2010.

[Jaz07] Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: 2007 Future of Software Engineering*, pages 199–213, Washington, DC, USA, 2007. IEEE Computer Society.

[Kae06] Stefan Kaes. Infoq: A look at common performance problems in rails. http://www.infoq.com/articles/Rails-Performance, June 2006. Accessed 15th June 2010.

[Kat09a] Yehuda Katz. Rails 3: Vaporware to awesome. Vaporware 2009, December 2009. Keynote Presentation.

[Kat09b] Yehuda Katz. Rails and merb merge: Performance (part 2 of 6) | engine yard blog. http://www.engineyard.com/blog/2009/rails-and-merb-merge-performance-part-2-of-6/, December 2009. Accessed 15th June 2010.

[Ken07] Josh Kenzer. 5 question interview with twitter developer alex payne | radical behavior. http://www.radicalbehavior.com/5-question-interview-with-twitter-developer-alex-payne/, March 2007. Accessed 15th June 2010.

[KG08] Wayne Kelly and John Gough. Ruby.net: a ruby compiler for the common language infrastructure. In *ACSC '08: Proceedings of the thirty-first Australasian conference on Computer science*, pages 37–46, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[KJZ02] Zbyszek P. Karkuszewski, Christopher Jarzynski, and Wojciech H. Zurek. Quantum chaotic environments, the butterfly effect, and decoherence. *Phys. Rev. Lett.*, 89(17):170405, Oct 2002.

[Kor06] Oded Koren. A study of the linux kernel evolution. *SIGOPS Oper. Syst. Rev.*, 40(2):110–112, 2006.

[Koz08] Marcus Koze. Quote on passenger for windows. http://izumi.plan99.net/blog/index.php/2008/04/11/phusion-passenger-mod_rails-public-release-heck-it's-about-time/#comment-8478, April 2008. Accessed 15th June 2010.

[Kru05] Steve Krug. *Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition)*. New Riders Publishing, Thousand Oaks, CA, USA, 2005.

[Lan] Jean-Philippe Lang. They are using redmine. http://www.redmine.org/wiki/redmine/TheyAreUsingRedmine. Accessed 15th June 2010.

[LB96] Kevin Lai and Mary Baker. A performance comparison of unix operating systems on the pentium. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 1996. USENIX Association.

[LCH+01] Wen-Syan Li, K. Selçuk Candan, Wang-Pin Hsiung, Oliver Po, and Divyakant Agrawal. Engineering high performance database-driven e-commerce web sites through dynamic content caching. In *EC-Web 2001: Proceedings of the Second International Conference on Electronic Commerce and Web Technologies*, pages 250–259, London, UK, 2001. Springer-Verlag.

[Lew06] Daniel Lewis. What is web 2.0? *Crossroads*, 13(1):3–3, 2006.

[Lin93] Mike Linksvayer. The choice of a gnu generation - an interview with linus torvalds. http://gondwanaland.com/meta/history/interview.html, 1993. Accessed 15th June 2010.

[Loh09] Eileen Loh. Jruby performance on glassfish v3 – part 1. http://weblogs.java.net/blog/eileeny/archive/2009/12/10/jruby-performance-glassfish-v3-part-1, December 2009. Accessed 15th June 2010.

[Lop] Brian Lopez. Readme: mysql2. http://github.com/brianmario/mysql2/blob/master/README.rdoc. Accessed 15th June 2010.

[Man] Ruby Version Manager. rvm + rubinius tiered benchmarks. http://rvm.beginrescueend.com/benchmarks/2010-01-06/. Accessed 15th June 2010.

[MGL+07] J. J. Merelo, Antonio Mora García, Juan Luis Jiménez Laredo, Juan Lupión, and Fernando Tricas. Browser-based distributed evolutionary computation: performance and scaling behavior. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2851–2858, New York, NY, USA, 2007. ACM.

[mmotRc] multiple members of the Ruby community. Module: Gc::profiler [rdoc documentation]. http://ruby-doc.org/ruby-1.9/classes/GC/Profiler.html. Accessed 15th June 2010.

[Mor02] Daniel L. Morrill. *Tuning and Customizing a Linux System*. APress L. P., 2002.

[MS05] Neil Matthew and Richard Stones. *Beginning Databases With Postgresql: From Novice To Professional (Beginning from Novice to Professional)*. Apress, Berkely, CA, USA, 2005.

[MT08] Tommi Mikkonen and Antero Taivalsaari. Web applications - spaghetti code for the 21st century. In *SERA '08: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society.

[Net] LTD. Netcraft. Most reliable hosting company sites in march 2010 - netcraft. http://news.netcraft.com/archives/2010/04/01/most_reliable_hosting_company_sites_in_march_2010.html. Accessed 15th June 2010.

[Neu08] Ryan Neufeld. Ruby oneliner: Benchmarking a string concordance. http://hammerofcode.com/category/general/, November 2008. Accessed 15th June 2010.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, SF, USA, 1993.

[NL06] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders Publishing, Thousand Oaks, CA, USA, 2006.

[NPD09] The NPD Group. The NPD Group: Despite recession, U.S. smartphone market is growing. http://www.npd.com/press/releases/press_090303.html, March 2009. Accessed 15th June 2010.

[NS06] Demetrius A. Nunes and Daniel Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 153–160, New York, NY, USA, 2006. ACM.

[Nun09] John Nunemaker. Why i think mongo is to databases what rails was to frameworks. http://goo.gl/to3U, December 2009. Accessed 15th June 2010.

[O'R] Tim O'Reilly. Quotes about ruby on rails. http://rubyonrails.org/quotes. Accessed 15th June 2010.

[O'R05] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. http://oreilly.com/web2/archive/what-is-web-20.html, September 2005. Accessed 15th June 2010.

[Phua] Phusion. Frequently asked questions - ruby enterprise edition. http://www.rubyenterpriseedition.com/faq.html#what_is_this. Accessed 15th June 2010.

[Phub] Phusion. Welcome - ruby enterprise edition. http://www.rubyenterpriseedition.com/. Accessed 15th June 2010.

[PRS01] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Beyond Interaction Design: Beyond Human-Computer Interaction*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[Rai] Rails Core Team. Ruby on rails 2.3 release notes. http://guides.rubyonrails.org/2_3_release_notes.html. Accessed 15th June 2010.

[RBH+95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 285–298, New York, NY, USA, 1995. ACM.

[Rob09] Ric Roberts. Getting started with mongodb and ruby. http://www.rubyinside.com/getting-started-mongodb-ruby-1875.html, June 2009. Accessed 15th June 2010.

[Rub] About Ruby. http://www.ruby-lang.org/en/about/. Accessed 15th June 2010.

[RYS09] Jayashree Ravi, Zhifeng Yu, and Weisong Shi. A survey on dynamic web content generation and delivery techniques. *J. Netw. Comput. Appl.*, 32(5):943–960, 2009.

[San04] Yohanes Santoso. Gnome's guide to webrick. October 2004.

[San10] Laurent Sansonetti. Macruby 0.6. http://www.macruby.org/blog/2010/04/30/macruby06.html, April 2010. Accessed 15th June 2010.

[Sas05] Koichi Sasada. Yarv: yet another rubyvm: innovating the ruby interpreter. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159, New York, NY, USA, 2005. ACM.

[Sch] Werner Schuster. New patches for 1.8.x fix memory leaks and improve performance. http://www.infoq.com/news/2009/01/ruby-patches-fix-leaks. Accessed 15th June 2010.

[Shaa] Zed A. Shaw. Mongrel faq. http://mongrel.rubyforge.org/wiki/FAQ. Accessed 15th June 2010.

[Shab] Zed A. Shaw. Ruby on rails: Core. http://unicorn.bogomips.org/. Accessed 15th June 2010.

[SHB06] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, 2006.

[sig] 37 signals. Unicorn: Rack http server. http://rubyonrails.org/core. Accessed 15th June 2010.

[Sil10] Gonçalo Silva. Scaling rails. volume 6, pages 9–10. Rails Magazine, 2010.

[SJW08] L.F.F. Stella, S. Jarzabek, and B. Wadhwa. A comparative study of maintainability of web applications on j2ee, .net and ruby on rails. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 93–99, Oct. 2008.

[Sla08] Michael Slater. The rebuilding and scaling of yellowpages.com. Rails Conf 2008, June 2008. Accessed 15th June 2010.

[SMB08] Sorin Stancu-Mara and Peter Baumann. A comparative benchmark of large objects in relational databases. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 277–284, New York, NY, USA, 2008. ACM.

[SN07] Elliot Smith and Rob Nichols. *Ruby on Rails Enterprise Application Development: Plan, Program, Extend Building a complete Ruby on Rails business application from start to finish*. Packt Publishing, 2007.

[Son09] Yuki Sonoda. Ruby 1.9.1 is released. http://groups.google.com/group/ruby-talk-google/browse_thread/thread/35e963933f9d0b1a, Jan 2009. Accessed 15th June 2010.

[Ste01] Bruce Stewart. An interview with the creator of ruby. http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html, November 2001. Accessed 15th June 2010.

[STT+09] Toyotaro Suzumura, Michiaki Tatsubori, Scott Trent, Akihiko Tozawa, and Tamiya Onodera. Highly scalable web applications with zero-copy data transfer. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 921–930, New York, NY, USA, 2009. ACM.

[SZT+08] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. *High performance mysql, 2nd edition*. O'Reilly, 2008.

REFERENCES

[TB06] Sun Microsystems Tim Bray. Issues in web frameworks, November 2006. Keynote Presentation.

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009.

[THB+06] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[TMIP08] Antero Taivalsaari, Tommi Mikkonen, Dan Ingalls, and Krzysztof Palacz. Web browser as an application platform: the lively kernel experience. Technical report, Mountain View, CA, USA, 2008.

[USA09] USA Today. Some PC makers don't know what to do with netbooks. http://www.usatoday.com/tech/products/2009-01-19-netbooks-future_N.htm?csp=34, January 2009. Accessed 15th June 2010.

[Vis08] Viswa Viswanathan. Rapid web application development: A ruby on rails tutorial. *IEEE Software*, 25:98–106, 2008.

[Wea09a] Evan Weaver. ree. http://blog.evanweaver.com/articles/2009/09/24/ree/, September 2009. Accessed 15th June 2010.

[Wea09b] Evan Weaver. ruby gc tuning. http://blog.evanweaver.com/articles/2009/04/09/ruby-gc-tuning/, April 2009. Accessed 15th June 2010.

[Wil07] Justin Williams. *Rails Solutions: Ruby on Rails Made Easy (Solutions)*. friends of ED, 2007.

[WWL05] Tongsen Wang, Lei Wu, and Zhangxi Lin. The revival of mozilla in the browser war against internet explorer. In *ICEC '05: Proceedings of the 7th international conference on Electronic commerce*, pages 159–166, New York, NY, USA, 2005. ACM.

[WZ09] Ye Diana Wang and Nima Zahadat. Teaching web development in the web 2.0 era. In *SIGITE '09: Proceedings of the 10th ACM conference on SIG-information technology education*, pages 80–86, New York, NY, USA, 2009. ACM.

[Yeh09] Yehuda Katz. Rails 3: The great decoupling. http://yehudakatz.com/2009/07/19/rails-3-the-great-decoupling/, July 2009. Accessed 15th June 2010.

[Yer09] John Yerhot. Extending rails through plugins. volume 1, page 3. Rails Magazine, 2009.

[YK94] Hyuck Yoo and Keng-Tai Ko. Operating system performance and large servers. In *EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop*, pages 166–171, New York, NY, USA, 1994. ACM.

[Zig06] Frank C. Ziglar. Comparing apache tomcat performance across platforms. http://www.webperformanceinc.com/library/reports/windows_vs_linux_part2/, March 2006. Accessed 15th June 2010.

REFERENCES

Appendix A

Ruby 1.9 Encoding Patch

The Ruby patch developed to fix the encoding issues when using non-ASCII characters on version 2.3.5 of Ruby on Rails.

```
# coding: UTF-8

# TODO: Most of these issues are not present in Rails 3. Remove this when updating.

# Force mysql rows to be UTF-8 (see rails.lighthouseapp.com/projects/8994/tickets/2476)
require 'mysql'

class Mysql::Result
  def encode(value, encoding = "utf-8")
    (String === value && value.respond_to?(:force_encoding)) ? value.force_encoding(encoding) : value
  end

  def each_utf8(&block)
    each_orig do |row|
      yield row.map {|col| encode(col) }
    end
  end
  alias each_orig each
  alias each each_utf8

  def each_hash_utf8(&block)
    each_hash_orig do |row|
      row.each {|k, v| row[k] = encode(v) }
      yield(row)
    end
  end
  alias each_hash_orig each_hash
  alias each_hash each_hash_utf8
end

# fix template rendering
module ActionView
  # NOTE: The template that this mixin is being included into is frozen
  # so you cannot set or modify any instance variables
  module Renderable #:nodoc:
```

```
36      extend ActiveSupport::Memoizable
37
38
39      private
40      def compile!(render_symbol, local_assigns)
41          locals_code = local_assigns.keys.map { |key| "#{key} = local_assigns
                [:#{key}];" }.join
42
43          source = <<-end_src
44            def #{render_symbol}(local_assigns)
45              old_output_buffer = output_buffer;#{locals_code};#{compiled_source}
46            ensure
47              self.output_buffer = old_output_buffer
48            end
49          end_src
50
51          # Workaround for erb
52          source.force_encoding('utf-8') if '1.9'.respond_to?(:force_encoding)
53
54          begin
55            ActionView::Base::CompiledTemplates.module_eval(source, filename, 0)
56          rescue Errno::ENOENT => e
57            raise e # Missing template file, re-raise for Base to rescue
58          rescue Exception => e # errors from template code
59            if logger = defined?(ActionController) && Base.logger
60              logger.debug "ERROR: compiling #{render_symbol} RAISED #{e}"
61              logger.debug "Function body: #{source}"
62              logger.debug "Backtrace: #{e.backtrace.join("\n")}"
63            end
64
65            raise ActionView::TemplateError.new(self, {}, e)
66          end
67        end
68
69    end
70  end
71
72  # the previous fix causes issues in uploaded files encoding, fixed here
73  module ActionController
74    class Request
75      private
76
77        # Convert nested Hashs to HashWithIndifferentAccess and replace
78        # file upload hashs with UploadedFile objects
79        def normalize_parameters(value)
80          case value
81          when Hash
82            if value.has_key?(:tempfile)
83              upload = value[:tempfile]
84              upload.extend(UploadedFile)
85              upload.original_path = value[:filename]
86              upload.content_type = value[:type]
87              upload
88            else
89              h = {}
90              value.each { |k, v| h[k] = normalize_parameters(v) }
91              h.with_indifferent_access
92            end
93          when Array
94            value.map { |e| normalize_parameters(e) }
```

```
95            else
96              value.force_encoding(Encoding::UTF_8) if value.respond_to?(:
                    force_encoding)
97              value
98            end
99          end
100       end
101     end
```

Appendix B

Ruby 1.9 Encoding Task

The rake task developed to automatically manage all Ruby files and set their default encoding to UTF-8. It inserts the encoding header when needed and standardizes the variant in use to "# coding: UTF-8".

```
1  desc "Manage the encoding header of Ruby files"
2  task :check_encoding_headers => :environment do
3    files = Array.new
4    ["*.rb", "*.rake"].each do |extension|
5      files.concat(Dir[ File.join(Dir.getwd.split(/\\/), "**", extension) ])
6    end
7
8    files.each do |file|
9      content = File.read(file)
10     next if content[0..16] == "# coding: UTF-8\n\n"
11
12     ["\n\n", "\n"].each do |file_end|
13       content = content.gsub(/(# encoding: UTF-8#{file_end})|(# coding: UTF-8#{
             file_end})|(# -*- coding: UTF-8 -*-#{file_end})/i, "")
14     end
15
16     new_file = File.open(file, "w")
17     new_file.write("# coding: UTF-8\n\n"+content)
18     new_file.close
19   end
20 end
```

Appendix C

Ruby 1.9 Configuration

The following command snippet starts the Ruby interpreter with an example configuration on UNIX.

```
1 export RUBY_HEAP_SLOTS_INCREMENT=500000
2 export RUBY_HEAP_MIN_SLOTS=500000
3 export RUBY_HEAP_SLOTS_GROWTH_FACTOR=1.1
4 export RUBY_GC_MALLOC_LIMIT=40000000
5 export RUBY_HEAP_FREE_MIN=100000
6 ruby
```

Appendix D

Ruby-Prof HTML Stack Printer

A snippet of the HTML output of this graphic hierarchical printer. It contains many information like the relative time elapsed of each call, the number of times a given call occured and even background colors to highlight faster and slower calls, among others.

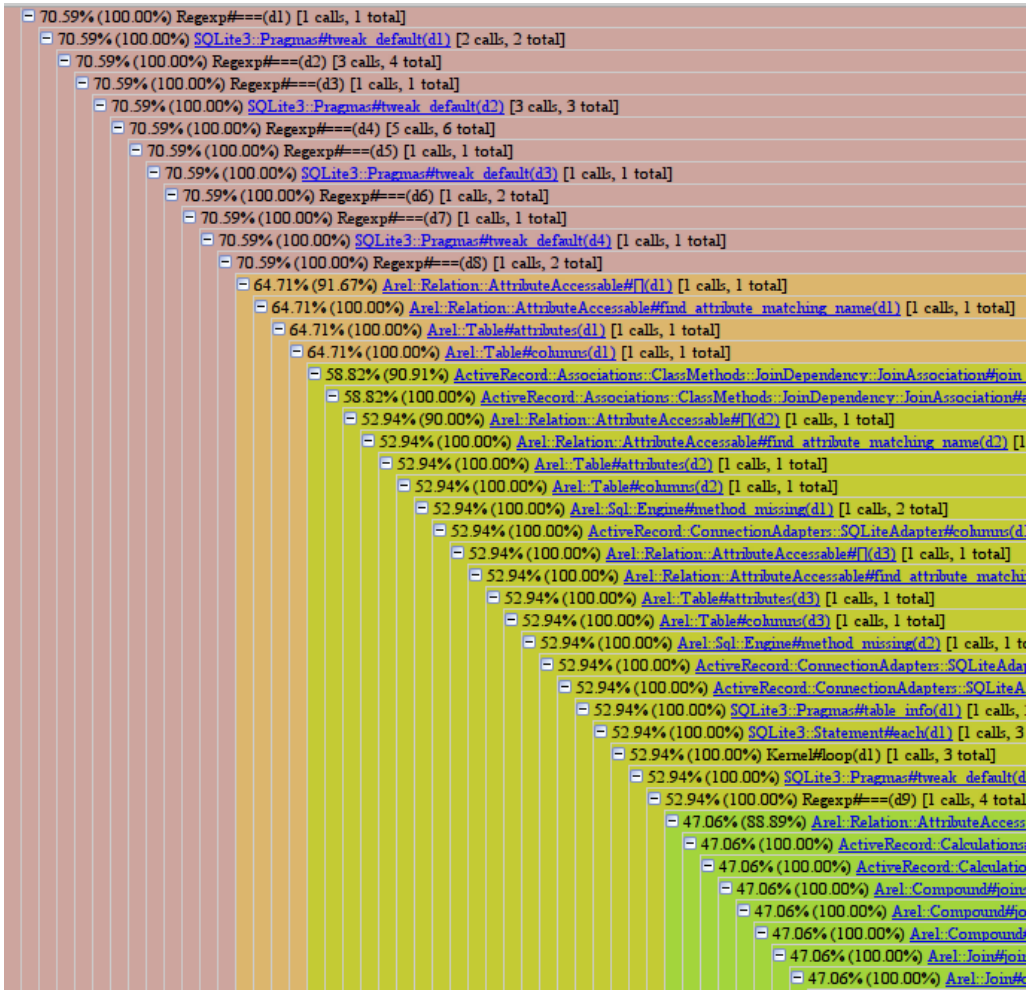

Figure D.1: Ruby-prof HTML Stack Printer

Appendix E

Memory Usage Monitor Script

A small Ruby script that measures the memory usage of specified processes. The user can configure the refresh interval and the output directory. The result is recorded in CSV.

```ruby
logs = Hash.new
temp_dir = ARGV[0]
delta = ARGV[1].to_f != 0 ? ARGV[1] : 1

while true do
  ARGV[1..-1].each do |arg|
    next if ARGV[1].to_f != 0
    (`pidof #{arg}`.split-logs.keys).each do |pid|
      logs[pid] = File.open("#{temp_dir}/#{(arg.gsub(/[^a-zA-Z 0-9]/, "")).gsub
          (/\s/,'-')}.#{pid}.csv", "w")
    end
  end

  logs.each do |pid, log|
    begin
      f = File.open("/proc/#{pid}/status", 'r').read
    rescue
      next
    end

    begin
      vmpeak = f.match('VmPeak:\s+(\d+)\s+kB')[1]
      vmsize = f.match('VmSize:\s+(\d+)\s+kB')[1]
      vmrss = f.match('VmRSS:\s+(\d+)\s+kB')[1]

      log.write("#{vmpeak},#{vmsize},#{vmrss}\n")
      log.flush
    rescue
      log.close
      logs.delete(pid)
    end
  end

  sleep delta.to_i
end
```

Appendix F

Lazy Type Casting in mysql2

A patch which enables lazy type casting of fields on of Ruby's *MySQL* database libraries, *mysql2*.

```
diff --git a/ext/mysql2_ext.c b/ext/mysql2_ext.c
index 3f791d3..fadea7b 100644
--- a/ext/mysql2_ext.c
+++ b/ext/mysql2_ext.c
@@ -370,7 +370,7 @@ static VALUE nogvl_read_query_result(void *ptr)
     return res == 0 ? Qtrue : Qfalse;
 }

-/* mysql_store_result may (unlikely) read rows off the socket */
+/*  may (unlikely) read rows off the socket */
 static VALUE nogvl_store_result(void *ptr)
 {
   MYSQL * client = ptr;
@@ -521,92 +521,8 @@ static VALUE rb_mysql_result_fetch_row(int argc, VALUE *argv, VALUE self) {
       }
       rb_ary_store(wrapper->fields, i, field);
     }
-
+    VALUE val = INT2NUM(wrapper->lastRowProcessed+i);
     if (row[i]) {
-      VALUE val;
-      switch(fields[i].type) {
-        case MYSQL_TYPE_NULL:       // NULL-type field
-          val = Qnil;
-          break;
-        case MYSQL_TYPE_BIT:        // BIT field (MySQL 5.0.3 and up)
-          val = rb_str_new(row[i], fieldLengths[i]);
-          break;
-        case MYSQL_TYPE_TINY:       // TINYINT field
-        case MYSQL_TYPE_SHORT:      // SMALLINT field
-        case MYSQL_TYPE_LONG:       // INTEGER field
-        case MYSQL_TYPE_INT24:      // MEDIUMINT field
-        case MYSQL_TYPE_LONGLONG:   // BIGINT field
-        case MYSQL_TYPE_YEAR:       // YEAR field
-          val = rb_cstr2inum(row[i], 10);
-          break;
-        case MYSQL_TYPE_DECIMAL:    // DECIMAL or NUMERIC field
```

```
38 -        case MYSQL_TYPE_NEWDECIMAL: // Precision math DECIMAL or NUMERIC field
    (MySQL 5.0.3 and up)
39 -          val = rb_funcall(cBigDecimal, intern_new, 1, rb_str_new(row[i],
    fieldLengths[i]));
40 -          break;
41 -        case MYSQL_TYPE_FLOAT:       // FLOAT field
42 -        case MYSQL_TYPE_DOUBLE:      // DOUBLE or REAL field
43 -          val = rb_float_new(strtod(row[i], NULL));
44 -          break;
45 -        case MYSQL_TYPE_TIME: {      // TIME field
46 -          int hour, min, sec, tokens;
47 -          tokens = sscanf(row[i], "%2d:%2d:%2d", &hour, &min, &sec);
48 -          val = rb_funcall(rb_cTime, intern_local, 6, INT2NUM(0), INT2NUM(1),
    INT2NUM(1), INT2NUM(hour), INT2NUM(min), INT2NUM(sec));
49 -          break;
50 -        }
51 -        case MYSQL_TYPE_TIMESTAMP:   // TIMESTAMP field
52 -        case MYSQL_TYPE_DATETIME: { // DATETIME field
53 -          int year, month, day, hour, min, sec, tokens;
54 -          tokens = sscanf(row[i], "%4d-%2d-%2d %2d:%2d:%2d", &year, &month, &
    day, &hour, &min, &sec);
55 -          if (year+month+day+hour+min+sec == 0) {
56 -            val = Qnil;
57 -          } else {
58 -            if (month < 1 || day < 1) {
59 -              rb_raise(cMysql2Error, "Invalid date: %s", row[i]);
60 -              val = Qnil;
61 -            } else {
62 -              val = rb_funcall(rb_cTime, intern_local, 6, INT2NUM(year),
    INT2NUM(month), INT2NUM(day), INT2NUM(hour), INT2NUM(min), INT2NUM(sec));
63 -            }
64 -          }
65 -          break;
66 -        }
67 -        case MYSQL_TYPE_DATE:        // DATE field
68 -        case MYSQL_TYPE_NEWDATE: {   // Newer const used > 5.0
69 -          int year, month, day, tokens;
70 -          tokens = sscanf(row[i], "%4d-%2d-%2d", &year, &month, &day);
71 -          if (year+month+day == 0) {
72 -            val = Qnil;
73 -          } else {
74 -            if (month < 1 || day < 1) {
75 -              rb_raise(cMysql2Error, "Invalid date: %s", row[i]);
76 -              val = Qnil;
77 -            } else {
78 -              val = rb_funcall(cDate, intern_new, 3, INT2NUM(year), INT2NUM(
    month), INT2NUM(day));
79 -            }
80 -          }
81 -          break;
82 -        }
83 -        case MYSQL_TYPE_TINY_BLOB:
84 -        case MYSQL_TYPE_MEDIUM_BLOB:
85 -        case MYSQL_TYPE_LONG_BLOB:
86 -        case MYSQL_TYPE_BLOB:
87 -        case MYSQL_TYPE_VAR_STRING:
88 -        case MYSQL_TYPE_VARCHAR:
89 -        case MYSQL_TYPE_STRING:      // CHAR or BINARY field
90 -        case MYSQL_TYPE_SET:         // SET field
91 -        case MYSQL_TYPE_ENUM:        // ENUM field
```

```
92 -          case MYSQL_TYPE_GEOMETRY:    // Spatial fielda
93 -          default:
94 -            val = rb_str_new(row[i], fieldLengths[i]);
95 -#ifdef HAVE_RUBY_ENCODING_H
96 -            // rudimentary check for binary content
97 -            if ((fields[i].flags & BINARY_FLAG) || fields[i].charsetnr == 63) {
98 -              rb_enc_associate_index(val, binaryEncoding);
99 -            } else {
100 -             rb_enc_associate_index(val, utf8Encoding);
101 -           }
102 -#endif
103 -           break;
104 -       }
105         rb_hash_aset(rowHash, field, val);
106       } else {
107         rb_hash_aset(rowHash, field, Qnil);
108 @@ -651,20 +567,20 @@ static VALUE rb_mysql_result_each(int argc, VALUE * argv,
      VALUE self) {
109           wrapper->lastRowProcessed++;
110         }
111
112 -       if (row == Qnil) {
113 +       /*if (row == Qnil) {
114           // we don't need the mysql C dataset around anymore, peace it
115           rb_mysql_result_free_result(wrapper);
116           return Qnil;
117 -       }
118 +       }*/
119
120         if (block != Qnil) {
121           rb_yield(row);
122         }
123       }
124 -     if (wrapper->lastRowProcessed == wrapper->numberOfRows) {
125 +     /*if (wrapper->lastRowProcessed == wrapper->numberOfRows) {
126         // we don't need the mysql C dataset around anymore, peace it
127         rb_mysql_result_free_result(wrapper);
128 -     }
129 +     }*/
130     }
131
132     return wrapper->rows;
133 @@ -686,6 +602,118 @@ static VALUE rb_raise_mysql2_error(MYSQL *client) {
134     return Qnil;
135   }
136
137 +static VALUE rb_mysql_result_cast(VALUE self, VALUE index) {
138 +   mysql2_result_wrapper * wrapper;
139 +   MYSQL_FIELD * field = NULL;
140 +   MYSQL_ROW row;
141 +   VALUE val;
142 +   unsigned long * fieldLengths;
143 +   void * ptr;
144 +
145 +   GetMysql2Result(self, wrapper);
146 +
147 +   if (wrapper->numberOfFields == 0) {
148 +     wrapper->numberOfFields = mysql_num_fields(wrapper->result);
149 +     wrapper->fields = rb_ary_new2(wrapper->numberOfFields);
150 +   }
```

```
151 +
152 +  unsigned long r = index / wrapper->numberOfRows;
153 +  int c = index % wrapper->numberOfFields;
154 +
155 +  ptr = wrapper->result;
156 +  mysql_data_seek(ptr, r); // <--- Segmentation fault
157 +  row = (MYSQL_ROW)rb_thread_blocking_region(nogvl_fetch_row, ptr, RUBY_UBF_IO
       , 0);
158 +
159 +  field = mysql_fetch_field_direct(ptr, c);
160 +  fieldLengths = mysql_fetch_lengths(wrapper->result);
161 +
162 +  switch(field->type) {
163 +    case MYSQL_TYPE_NULL:       // NULL-type field
164 +      val = Qnil;
165 +      break;
166 +    case MYSQL_TYPE_BIT:        // BIT field (MySQL 5.0.3 and up)
167 +      val = rb_str_new(row[c], fieldLengths[c]);
168 +      break;
169 +    case MYSQL_TYPE_TINY:       // TINYINT field
170 +    case MYSQL_TYPE_SHORT:      // SMALLINT field
171 +    case MYSQL_TYPE_LONG:       // INTEGER field
172 +    case MYSQL_TYPE_INT24:      // MEDIUMINT field
173 +    case MYSQL_TYPE_LONGLONG:   // BIGINT field
174 +    case MYSQL_TYPE_YEAR:       // YEAR field
175 +      val = rb_cstr2inum(row[c], 10);
176 +      break;
177 +    case MYSQL_TYPE_DECIMAL:    // DECIMAL or NUMERIC field
178 +    case MYSQL_TYPE_NEWDECIMAL: // Precision math DECIMAL or NUMERIC field (
       MySQL 5.0.3 and up)
179 +      val = rb_funcall(cBigDecimal, intern_new, 1, rb_str_new(row[c],
       fieldLengths[c]));
180 +      break;
181 +    case MYSQL_TYPE_FLOAT:      // FLOAT field
182 +    case MYSQL_TYPE_DOUBLE:     // DOUBLE or REAL field
183 +      val = rb_float_new(strtod(row[c], NULL));
184 +      break;
185 +    case MYSQL_TYPE_TIME: {     // TIME field
186 +      int hour, min, sec, tokens;
187 +      tokens = sscanf(row[c], "%2d:%2d:%2d", &hour, &min, &sec);
188 +      val = rb_funcall(rb_cTime, intern_local, 6, INT2NUM(0), INT2NUM(1),
       INT2NUM(1), INT2NUM(hour), INT2NUM(min), INT2NUM(sec));
189 +      break;
190 +    }
191 +    case MYSQL_TYPE_TIMESTAMP:  // TIMESTAMP field
192 +    case MYSQL_TYPE_DATETIME: { // DATETIME field
193 +      int year, month, day, hour, min, sec, tokens;
194 +      tokens = sscanf(row[c], "%4d-%2d-%2d %2d:%2d:%2d", &year, &month, &day,
       &hour, &min, &sec);
195 +      if (year+month+day+hour+min+sec == 0) {
196 +        val = Qnil;
197 +      } else {
198 +        if (month < 1 || day < 1) {
199 +          rb_raise(cMysql2Error, "Invalid date: %s", row[c]);
200 +          val = Qnil;
201 +        } else {
202 +          val = rb_funcall(rb_cTime, intern_local, 6, INT2NUM(year), INT2NUM(
       month), INT2NUM(day), INT2NUM(hour), INT2NUM(min), INT2NUM(sec));
203 +        }
204 +      }
```

```
205 +       break;
206 +     }
207 +     case MYSQL_TYPE_DATE:         // DATE field
208 +     case MYSQL_TYPE_NEWDATE: {    // Newer const used > 5.0
209 +       int year, month, day, tokens;
210 +       tokens = sscanf(row[c], "%4d-%2d-%2d", &year, &month, &day);
211 +       if (year+month+day == 0) {
212 +         val = Qnil;
213 +       } else {
214 +         if (month < 1 || day < 1) {
215 +           rb_raise(cMysql2Error, "Invalid date: %s", row[c]);
216 +           val = Qnil;
217 +         } else {
218 +           val = rb_funcall(cDate, intern_new, 3, INT2NUM(year), INT2NUM(month)
      , INT2NUM(day));
219 +         }
220 +       }
221 +       break;
222 +     }
223 +     case MYSQL_TYPE_TINY_BLOB:
224 +     case MYSQL_TYPE_MEDIUM_BLOB:
225 +     case MYSQL_TYPE_LONG_BLOB:
226 +     case MYSQL_TYPE_BLOB:
227 +     case MYSQL_TYPE_VAR_STRING:
228 +     case MYSQL_TYPE_VARCHAR:
229 +     case MYSQL_TYPE_STRING:       // CHAR or BINARY field
230 +     case MYSQL_TYPE_SET:          // SET field
231 +     case MYSQL_TYPE_ENUM:         // ENUM field
232 +     case MYSQL_TYPE_GEOMETRY:     // Spatial fielda
233 +     default:
234 +       val = rb_str_new(row[c], fieldLengths[c]);
235 +#ifdef HAVE_RUBY_ENCODING_H
236 +       // rudimentary check for binary content
237 +       if ((field->flags & BINARY_FLAG) || field->charsetnr == 63) {
238 +         rb_enc_associate_index(val, binaryEncoding);
239 +       } else {
240 +         rb_enc_associate_index(val, utf8Encoding);
241 +       }
242 +#endif
243 +       break;
244 +   }
245 +
246 +   return val;
247 +}
248 +
249  /* Ruby Extension initializer */
250  void Init_mysql2_ext() {
251    rb_require("date");
252 @@ -709,6 +737,7 @@ void Init_mysql2_ext() {
253    rb_define_method(cMysql2Client, "async_result", rb_mysql_client_async_result
          , 0);
254    rb_define_method(cMysql2Client, "last_id", rb_mysql_client_last_id, 0);
255    rb_define_method(cMysql2Client, "affected_rows",
          rb_mysql_client_affected_rows, 0);
256 +  rb_define_method(cMysql2Client, "cast", rb_mysql_result_cast, 1);
257
258    cMysql2Error = rb_define_class_under(mMysql2, "Error", rb_eStandardError);
259    rb_define_method(cMysql2Error, "error_number", rb_mysql_error_error_number,
          0);
260 @@ -716,6 +745,9 @@ void Init_mysql2_ext() {
```

```
261
262     cMysql2Result = rb_define_class_under(mMysql2, "Result", rb_cObject);
263     rb_define_method(cMysql2Result, "each", rb_mysql_result_each, -1);
264 +
265 +   //cMysql2Type = rb_define_class_under(mMysql2, "Type", rb_cObject);
266 +   //rb_define_singleton_method(cMysql2Type, "cast", rb_mysql_result_cast, 1);
267
268     VALUE mEnumerable = rb_const_get(rb_cObject, rb_intern("Enumerable"));
269     rb_include_module(cMysql2Result, mEnumerable);
270 diff --git a/lib/active_record/connection_adapters/mysql2_adapter.rb b/lib/
      active_record/connection_adapters/mysql2_adapter.rb
271 index 825dd44..7daa537 100644
272 --- a/lib/active_record/connection_adapters/mysql2_adapter.rb
273 +++ b/lib/active_record/connection_adapters/mysql2_adapter.rb
274 @@ -14,6 +14,12 @@ module ActiveRecord
275     module ConnectionAdapters
276       class Mysql2Column < Column
277         BOOL = "tinyint(1)".freeze
278 +
279 +       def initialize(name, default, connection, sql_type = nil, null = true)
280 +         @connection = connection
281 +         super(name, default, sql_type, null)
282 +       end
283 +
284         def extract_default(default)
285           if sql_type =~ /blob/i || type == :text
286             if default.blank?
287 @@ -48,43 +54,47 @@ module ActiveRecord
288             when :boolean         then Object
289           end
290         end
291 -
292 +
293 +       # Gets the value (from the index)
294         def type_cast(value)
295           return nil if value.nil?
296           case type
297 -           when :string                then value
298 -           when :text                  then value
299 -           when :integer               then value.to_i rescue value ? 1 : 0
300 -           when :float                 then value.to_f # returns self if it's
      already a Float
301 -           when :decimal               then self.class.value_to_decimal(value)
302 -           when :datetime, :timestamp  then value.class == Time ? value : self.
      class.string_to_time(value)
303 -           when :time                  then value.class == Time ? value : self.
      class.string_to_dummy_time(value)
304 -           when :date                  then value.class == Date ? value : self.
      class.string_to_date(value)
305 -           when :binary                then value
306 -           when :boolean               then self.class.value_to_boolean(value)
307 -           else value
308 +           when :string    then Mysql2::Type.cast(value)
309 +           when :text      then Mysql2::Type.cast(value)
310 +           when :integer   then Mysql2::Type.cast(value).to_i rescue Mysql2::
      Type.cast(value) ? 1 : 0
311 +           when :float     then Mysql2::Type.cast(value).to_f
312 +           when :decimal   then self.class.value_to_decimal(Mysql2::Type.cast(
      value))
```

```
313 +           when :datetime  then self.class.string_to_time(Mysql2::Type.cast(
    value))
314 +           when :timestamp then self.class.string_to_time(Mysql2::Type.cast(
    value))
315 +           when :time      then self.class.string_to_dummy_time(Mysql2::Type.
    cast(value))
316 +           when :date      then self.class.string_to_date(Mysql2::Type.cast(
    value))
317 +           when :binary    then self.class.binary_to_string(Mysql2::Type.cast(
    value))
318 +           when :boolean   then self.class.value_to_boolean(Mysql2::Type.cast(
    value))
319 +           else Mysql2::Type.cast(value)
320           end
321         end
322
323         def type_cast_code(var_name)
324           case type
325 -           when :string                then nil
326 -           when :text                  then nil
327 -           when :integer               then "#{var_name}.to_i rescue #{var_name
    } ? 1 : 0"
328 -           when :float                 then "#{var_name}.to_f"
329 -           when :decimal               then "#{self.class.name}.
    value_to_decimal(#{var_name})"
330 -           when :datetime, :timestamp  then "#{var_name}.class == Time ? #{
    var_name} : #{self.class.name}.string_to_time(#{var_name})"
331 -           when :time                  then "#{var_name}.class == Time ? #{
    var_name} : #{self.class.name}.string_to_dummy_time(#{var_name})"
332 -           when :date                  then "#{var_name}.class == Date ? #{
    var_name} : #{self.class.name}.string_to_date(#{var_name})"
333 -           when :binary                then nil
334 -           when :boolean               then "#{self.class.name}.
    value_to_boolean(#{var_name})"
335 +           when :string    then "Mysql2::Type.cast(#{var_name})"
336 +           when :text      then "Mysql2::Type.cast(#{var_name})"
337 +           when :integer   then "(Mysql2::Type.cast(#{var_name}).to_i rescue
    Mysql2::Type.cast(#{var_name}) ? 1 : 0)"
338 +           when :float     then "Mysql2::Type.cast(#{var_name}).to_f"
339 +           when :decimal   then "#{self.class.name}.value_to_decimal(Mysql2::
    Type.cast(#{var_name}))"
340 +           when :datetime  then "#{self.class.name}.string_to_time(Mysql2::Type
    .cast(#{var_name}))"
341 +           when :timestamp then "#{self.class.name}.string_to_time(Mysql2::Type
    .cast(#{var_name}))"
342 +           when :time      then "#{self.class.name}.string_to_dummy_time(Mysql2
    ::Type.cast(#{var_name}))"
343 +           when :date      then "#{self.class.name}.string_to_date(Mysql2::Type
    .cast(#{var_name}))"
344 +           when :binary    then "#{self.class.name}.binary_to_string(Mysql2::
    Type.cast(#{var_name}))"
345 +           when :boolean   then "#{self.class.name}.value_to_boolean(Mysql2::
    Type.cast(#{var_name}))"
346           else nil
347           end
348         end
349
350         private
351           def simplified_type(field_type)
```

```
352 -           return :boolean if Mysql2Adapter.emulate_booleans && field_type.
    downcase.index(BOOL)
353 +           puts field_type
354 +           return :boolean if Mysql2Adapter.emulate_booleans && @connection.
    cast(field_type).downcase.index(BOOL)
355             return :string  if field_type =~ /enum/i
356             return :integer if field_type =~ /year/i
357             super
358 @@ -414,7 +424,7 @@ module ActiveRecord
359           columns = []
360           result = execute(sql, :skip_logging)
361           result.each(:symbolize_keys => true) { |field|
362 -           columns << Mysql2Column.new(field[:Field], field[:Default], field[:
    Type], field[:Null] == "YES")
363 +           columns << Mysql2Column.new(field[:Field], field[:Default],
    @connection, field[:Type], field[:Null] == "YES")
364           }
365           columns
366         end
367 @@ -592,4 +602,4 @@ module ActiveRecord
368           end
369       end
370     end
371 -end
372 \ No newline at end of file
373 +end
```

Appendix G

"Scaling Rails" Article on Rails Magazine

Scaling Rails
by Gonçalo Silva

The term Web 2.0, born somewhere in 2001, is related to improving the first version of the Web. It aims at improving user experiences, by providing better usability and more dynamic content. Many web frameworks, including Ruby on Rails, were born as part of this huge web momentum that we still live in nowadays.

Most websites are built to provide great user experiences. Recent studies show that users won't wait longer than 8 seconds before leaving a slow or unresponsive website. This value keeps getting lower and lower as users become more demanding.

This is the introductory article in a short series related to Ruby on Rails Scalability and Performance Optimization. This great framework's performance is conditioned by every related component, besides itself.

G.1 Website Performance

Developers often oversee that web applications need to be fast and very responsive as part of a richer user experience. Having a highly efficient platform will generally allow lower expenses on hardware but also lower response times, making its users happier. In some cases this need can be extreme—a high-demand platform strives for scalability as its users keep growing.

Ruby on Rails is widely known for being optimized for programmer productivity and happiness, but its scalability or performance are not generally favored. Many well-known platforms like Twitter or Scribd have put enormous efforts in improving these characteristics and sometimes faced a few issues while doing it—we all know the famous "fail whale".

G.2 System Resources

Very little people have access to topnotch resources. Most need to deploy and maintain a Rails application on a shared host, limited VPS or even a dedicated server. High-end computers or server clusters are not easily accessible by the masses but every developer should to be able to provide great services with limited resources.

The answer to every scalability issue is not "just throw hardware at it".

G.3 Involved Components

Most servers run Linux, while a few use FreeBSD. Every operating system has a different philosophy to almost everything – from the file system to networking I/O, and these details can impact all the other components.

Some applications use Ruby 1.8, while others are already riding Ruby 1.9. There are many Ruby interpreters, from MRI to YARV, including widely-known implementations like Ruby Enterprise Edition, JRuby or Rubinius. Each of these has its particular characteristics, offering different advantages and disadvantages.

Very few applications are built or have been ported to Rails 3, which suffered huge improvements on performance. Porting is an important step as Rails 3 provides reduced computing times and memory usage, when compared to its predecessor.

When it comes to web servers, the number of choices are tremendous. From the old combination of Apache and Mongrel, or the famous Passenger for Apache and Nginx, or even some newcomers like Thin and Unicorn, every application has an ideal setup and its web server architecture greatly impacts its performance and memory usage.

The database choices, ranging from popular relational databases like MySQL to more recent projects like Cassandra or MongoDB are also associated with strong advantages and disadvantages. This aspect gains even greater importance when considering that the database is a major performance bottleneck.

Let's not forget about the application itself: a few coding conventions could be followed to improve the application's performance, scalability and, most importantly, the code's quality.

Your system, your architecture. It's all about choice.

Appendix H

General Guidelines and Conventions for Optimizing Rails Applications

Building highly performant Ruby on Rails applications involves carefully examining all system components. The following sections will briefly analyze each component, exhibiting its alternatives and analyzing their benefits and shortcomings when compared to other possible solutions.

H.1 Operating System

From a performance perspective, Linux is the best operating system to use when deploying Rails applications. Comparing with other operating systems, MRI is:

- approximately 100% faster than its Windows counterpart;
- approximately 30% faster than its BSD counterpart.

On a similar tone, YARV on Linux is:

- approximately 70% faster than its Windows counterpart;
- approximately 22% faster than its BSD counterpart.

MySQL also yields better performance in UNIX systems, since Windows restrains its concurrent capabilities (by limiting the allowed number of opened files).

Gentoo is the most stable and scalable Linux distribution. It was able to scale up to 10000 concurrent requests on a simple page responding on an acceptable time span while Debian, Ubuntu and CentOS were not.

Regarding server configuration the kernel should be using the Deadline I/O scheduler, a non-preemptive configuration and a timer interrupt of 100Hz. Its sysctl configuration should also be changed to improve its scalability and stability, namely:

- net.core.rmem_max — 16777216
- net.core.wmem_max — 16777216
- net.ipv4.tcp_rmem — 4096 87380 16777216
- net.ipv4.tcp_wmem — 4096 87380 16777216

- net.core.netdev_max_backlog — 4096
- net.core.somaxconn — 4096
- net.ipv4.tcp_tw_reuse — 1
- net.ipv4.tcp_tw_recycle — 1
- net.ipv4.tcp_fin_timeout — 15
- net.ipv4.tcp_timestamps — 0
- net.ipv4.tcp_orphan_retries — 1

H.2 Ruby

YARV is the Ruby interpreter with the best overall performance. Applications still using Ruby 1.8 can be easily ported to the new version and this upgrade is highly advisable. However, for those applications that have specific Ruby 1.8-related requirements, the best performing interpreter is JRuby.

Regarding YARV, there is a fork on GitHub which enables parameter configuration for adaptive performance at http://github.com/goncalossilva/ruby. To run Ruby with customized configurations, create the following script:

```
1 export RUBY_HEAP_SLOTS_INCREMENT=500000
2 export RUBY_HEAP_MIN_SLOTS=500000
3 export RUBY_HEAP_SLOTS_GROWTH_FACTOR=1.1
4 export RUBY_GC_MALLOC_LIMIT=40000000
5 export RUBY_HEAP_FREE_MIN=100000
6 ruby
```

The supported parameters are the previously exhibited. Each addresses a different functionality:

- RUBY_HEAP_SLOTS_INCREMENT —Initial number of heap slots. It also represents the minimum number of slots, at all times (default: 10000);
- RUBY_HEAP_MIN_SLOTS — The number of new slots to allocate when all initial slots are used (default: 10000);
- RUBY_HEAP_SLOTS_GROWTH_FACTOR — Next time Ruby needs new heap slots it will use a multiplicator, defined by this environment variable's value (default: 1.8, meaning it will allocate 18000 new slots if default settings are in use);
- RUBY_GC_MALLOC_LIMIT — The number of C data structures that can be allocated before triggering the garbage collector. This one is very important since the default value makes the GC run when there are still empty heap slots because Rails allocates and deallocates a lot of data (default: 8000000);
- RUBY_HEAP_FREE_MIN — The number of free slots that should be present after GC finishes running. If there are fewer slots than those defined it will allocate new ones according to the value of RUBY_HEAP_SLOTS_INCREMENT and the previously mentioned value of RUBY_HEAP_SLOTS_GROWTH_FACTOR (default: 4096).

Each application has its own optimal configuration. Twitter's and 37signals' settings are shown bellow:

```
# 37signals
RUBY_HEAP_MIN_SLOTS=600000
RUBY_GC_MALLOC_LIMIT=59000000
RUBY_HEAP_FREE_MIN=100000

# Twitter
RUBY_HEAP_MIN_SLOTS=500000
RUBY_HEAP_SLOTS_INCREMENT=250000
RUBY_HEAP_SLOTS_GROWTH_FACTOR=1
RUBY_GC_MALLOC_LIMIT=50000000
```

Developers should benchmark their applications to find an optimal value combination for the best performance results.

H.3 Web Servers

Thin, Unicorn and Passenger have very similar performances. However, Thin uses slightly less memory and accomplishes similar results.

Thin is optimized for fast clients so it should be used in combination with a reverse proxy which buffers requests/replies for slow clients. Nginx is the optimal choice for its improved performance, stability and memory usage.

Thin as an option to enable threading. This, however, generally decreases this web server's performance, so the advice is to avoid using it. Nginx, on the other hand, has a powerful configuration option since it allows developers to specify the event model to use. One must use the model optimized for the running operating system which, for Linux, is *epoll*.

H.4 Databases

MySQL is the best performing database among relation databases. Non-relational solutions, however, trade enhanced read/write speeds for higher disk usage. MongoDB seems to be the best performing non-relational database at this moment. It is up to the developer to decide which type of database to use, considering the mentioned benefits and shortcomings.

In any case, *caching* is very important since it generally significantly improves the database performance.

If an application is using MySQL, alternate Ruby database libraries should be considered, namely *mysql2* or *mysqlplus*. Both perform better than the default one, providing significant improvements in database interactions.

H.5 Ruby on Rails

Rails 3 is remarkably faster than Rails 2 and it is nearing its first stable release. Upon commencing new projects, this version should be carefully considered.

Despite the version in use, there are some common Rails' features which can significantly help improving an applications' performance, namely:

- Eager loading should be used whenever a record and any number of associations are being fetched from the database;
- Transactions should be used to wrap consecutive writes to the database, on either CREATE and UPDATE statements;
- Magic finders should be generally avoidable. Normal finds are generally very readable, so improving their readability will not likely be worthy given the performance penalty involved;
- Fetching large groups of records should be done using "find_each" or "find_in_batches" and not using the regular "find" method.

When paired with Nginx, Rails can benefit from its *X-Accel-Redirect* feature on file downloads by using the plugin found at `http://github.com/goncalossilva/x-accel-redirect`.

Finally, profiling is the most important part of fine-tunning. The current Ruby/Rails profiling tools integrate seamlessly and support many data visualization types, including an hierarchical HTML call stack very useful for non-automated analysis. All these tools should be used to spot the existing bottlenecks in the application itself.