BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE TESTING OF LINUX KERNEL SCHEDULER

VÝKONNOSTNÍ TESTOVÁNÍ PLÁNOVAČE LINUXOVÉHO KERNELU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
JIŘÍ VOZÁR

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. VIKTOR MALÍK

BRNO 2019

Department of Intelligent Systems (DITS) Academic year 2018/2019

Bachelor's Thesis Specification

21469

Student:	**Vozár Jiří**
Programme:	Information Technology
Title:	**Performance Testing of Linux Kernel Scheduler**
Category:	Software analysis and testing

Assignment:

1. Get acquainted with the existing methods for measuring performance of the Linux kernel scheduler and with means of storing of benchmarks results for further processing.
2. Study possible ways of processing these results with a focus on graphic interpretation and on methods for detection of performance degradation.
3. Design and implement a method for efficient graphic interpretation of long-term measurements.
4. Design and implement a method for automatic detection of performance regression.
5. Demonstrate the functionality of your implementation on at least two versions of the Linux kernel.
6. Evaluate the obtained results and discuss possibilities of further development of the project, especially of the automatic detection of performance regression.

Recommended literature:

- Lozi, Jean-Pierre, et al. "The Linux scheduler: a decade of wasted cores." *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.
- Daniel, P., and Cesati Marco. "Understanding the Linux kernel." (2007).
- Bailey, David H., et al. "The NAS parallel benchmarks." *The International Journal of Supercomputing Applications* 5.3 (1991): 63-73.

Requirements for the first semester:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Malík Viktor, Ing.**
Consultant:	Tišnovský Pavel, Ing., Ph.D., RedHatCZ
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 15, 2019
Approval date:	November 1, 2018

Abstract

Performance of process scheduler in a kernel of an operating system significantly influences throughput and latency of all applications running above it. Any performance drop can have critical consequences on the applications. With the arrival of every new technology (e.g. symetric multiprocesing) the code of the scheduler evolves and grows. This requires not only functional, but also performance regression testing. This work presents methods of performance testing used in the Red Hat, Inc. company. It describes how one can measure performance of the Linux process scheduler in the Linux kernel, collect statistics about its behavior, store the collected data, and visualize them. The goal of this work is to design and implement a new technique of visualization of long-term measurements and utilization of machine learning for automatic classification of performance degradation between different results.

Abstrakt

Výkon plánovače procesů v jádře operačního systému značně ovlivňuje rychlost a odezvu všech aplikací, které na něm běží. Jakýkoli propad výkonu pak může mít kritické důsledky na běhu aplikací. S příchodem každé nové technologie (např. symetrický multiprocesing) se kód plánovače vyvíjí a rozšiřuje. Proto jsou potřeba regresní testy nejen na jeho fukčnost, ale i výkon. Tato práce mapuje metody testování plánovače operačního systému Linux ve firmě Red Hat. Popisuje způsoby měření výkonu plánovače, sbírání informací o jeho chování, ukládání sesbíraných dat a jejich vizualizaci. Hlavním cílem práce je pak návrh a implementace nového způsobu vizualizace dlouhodobých měření a využití strojového učení pro automatické rozpoznání degradace výkonu mezi dvěma výsledky.

Keywords

Linux, kernel, task scheduler, CFS, testing, performance measurement, visualization, machine learning

Klíčová slova

Linux, jádro, plánovač úloh, CFS, testování, měření výkonu, vizualiace, strojové učení

Reference

VOZÁR, Jiří. *Performance Testing of Linux Kernel Scheduler.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

Plánovač procesů v operačním systému se stará o přidělování procesorového času běžícím procesům a jejich rovnoměrné rozložení mezi procesorové jádra. Výkon plánovače procesů pak silně ovlivňuje i výkon samotných aplikací běžících na daném operačním systému. Jakýkoli propad výkonu pak může mít v komerční sféře za následky vysoké finanční ztráty.

Plánovač procesů funguje velmi jednoduše pro systémy s jedním jádrem, ovšem s příchodem vícejádrových procesorů se plánování zkomplikovalo vyvažováním front procesů mezi jádry a trvalo, než se všechny problémy vyladily. Nové komplikace plánování pak přinesly víceprocesorové systémy s neuniformní dobou přístupu do paměti. Plánování na této architektuře je stále ve vývoji a tím i náchylné k chybám způsobujícím propad výkonu. Proto je potřeba regresní testování výkonu nových verzí jádra pro včasné odhalení těchto chyb.

Na rozdíl od funkčního testování není výkonové testování není jednoznačný výsledek, jestli test doběhl. Pro zjištění změny výkonu je potřeba relativní porovnání naměřených hodnot s referenčním výsledkem z předchozí verze nebo jiné konfigurace a určení prahu mezi odchylkou měření a skutečným propadem výkonu. Propady výkonu plánovače navíc nejsou způsobeny pomalým kódem, ale chybným přemisťováním procesů a jejich dat mezi jádry a fyzickými procesory.

Tato práce popisuje výkonové testování plánovače procesů Red Hat Enterprise Linuxu ve společnosti Red Hat, Inc. Běžný způsob zjišťování výkonu plánovače je měření benchmarkem, který simuluje reálnou zátěž. Benchmarků je více s různými způsoby zátěže, především však pomocí většího množství procesů nebo vláken s vnitřní komunikací. Výsledky jsou pak systematicky ukládány pro snažší tvorbu porovnání výkonu. Pro efektivní analýzu výkonu je pak nutná volba správné vizualizace pro rychlé odhalení zdroje problému.

Práce navrhuje a implementuje novou metodu zobrazování dlouhodobých výsledků měření výkonu nazvanou *timelines*. Výstupem je pak HTML stránka obsahující krabicové (box plot) grafy a shrnující tabulky znázorňující změnu výkonu a přesnost měření v průběhu verzí jádra operačního systému. Tento výstup již pomohl s redukcí nestabilních částí benchmarků a průběžně slouží pro sledování změn výkonu a dohledávání verzí, kde výkon změnil.

Dále práce navrhuje použití strojového učení pro automatickou klasifikaci porovnání dvou výsledků, zda se mezi nimi i přes nepřesnost měření a šum projevuje propad výkonu. Práce navrhuje předzpracování výsledků měření pro učení klasifikátorů a porovnává různé klasifikátory pro budoucí začlenění do generátoru zpráv s porovnáním nových výsledků pro urychlení jejich analýzy.

Performance Testing of Linux Kernel Scheduler

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Viktor Malík. The supplementary information was provided by Ing. Pavel Tišnovský, Ph.D.from Red Hat Czech s.r.o. and Ing. Tomáš Fiedor. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Jiří Vozár
May 14, 2019

Acknowledgements

I would like to thank to my supervisor Ing. Viktor Malík and consultants Ing. Pavel Tišnovský, Ph.D.from Red Hat Czech s.r.o. and Ing. Tomáš Fiedor for for guidance, useful revisions of the text and consultations. Also I would like to thank my manager RNDr. Jiří Hladký and collegue Bc. Kamil Kolakowski for insight to scheduler testing and ideas for proposed work.

Contents

Chapter 1

Introduction

Performance of an operating system is crucial as any triggered degradation can significantly affect the performance of all applications running above it. Moreover, when a new version is released and it introduces a performance regression, it can break the stability of e.g. business applications leading to great financial losses. An important part of operating system and the main influence on its performance is the implementation and strategy of a process scheduler, which manages processes and their processor time.

In the Linux kernel, scheduler used to be simple, but with an introduction of multi-core CPUs achieving stable performance required adapting multiple runqueues for each core and their balancing. This lead to complex code and it took some time to eliminate the bugs and tune the scheduler's behavior. Another change came with symmetric multiprocessing technology bringing to market machines with multiple CPU sockets and non-uniform memory organization and access [8]. Scheduling on such systems is still under an active development and therefore it is even more prone to performance degradation bugs. In order to discover these bugs and to keep the performance of the operating system stable, using performance testing was essential. Due to this fact, the performance of the scheduler has to be evaluated before each release.

Compared to functional testing, performance cannot be evaluated as just true and false results. It is more challenging as we have to (1) compare it relatively with previous versions or measurements, and (2) choose a suitable threshold when reporting the performance regression. Due to complexity of the scheduler, the common tools for inspecting performance may often yield unsatisfying results. Moreover, the biggest performance regressions of the scheduler do not dwell in an inefficient code, but instead in an inefficient assignment of processes to the CPU cores and their queues.

This thesis focuses on performance testing of the Red Hat Enterprise Linux (RHEL) kernel scheduler managed by the Red Hat, Inc. company. The usual performance testing method of the scheduler is to simulate load similar to the real usage. Currently, there are many benchmarks targeting different types of load, usually spawning many parallel process, sometimes even communicating between each other. The results of measurements of the performance must be stored systematically and effectively for later comparisons. So in order to effectively interpret the collected results and their comparisons, their effective visualization is essential.

In this thesis, we propose a new interpretation of long-term results of the benchmarks using box plot graphs in order to enhance the process of data analysis. This visualization should help with inspecting the measurement stability of benchmarks and with finding versions where performance degradations appeared or were fixed.

Moreover, to reduce time one has to spend analyzing these comparisons, we propose an utilization of machine learning methods that will automatically check for possible degradations in the Linux kernel scheduler. We will describe how to create the dataset for classification, compare different classifiers, and evaluate their accuracy on the dataset.

This thesis is structured as follows. In Chapter 2 we describe Completely Fair Scheduler – the current process scheduler of the Linux kernel, architecture of symmetric multiprocessing systems, and how the scheduler handles them. In Chapter 3, we describe the performance measurement of the Linux process scheduler. First we introduce benchmarks used in Red Hat for load testing of the scheduler performance and then we describe complementary tools for analysis of behavior of the scheduler and of the system. Ways of storage of the collected data and their visualization for comparison are described in Chapter 4.

We propose a new way of visualization of long-term results comparison called *Timelines* in Chapter 5. In Chapter 6, we propose utilization of machine learning methods for automatic classification of comparison of two results to recognize a performance regression.

Chapter 2

Linux process scheduling

Process scheduler is a part of an operating system which assigns the processor time to tasks. Its main goal is to maximize effectivity of the CPU usage and to ensure fairness of the CPU time assigned to each task.

There are two opposing targets for a scheduler: either maximizing the throughput or minimizing the latency. Lower amount of context switches leaves more CPU time for tasks, but raises the response time on system events. While users' workstations aims for a low response time, computational servers require high throughput. Scheduler can then be usually tuned to fit the intended purpose.

In this chapter we describe basic behavior of Linux process scheduler. Then we compare uniform and non-uniform memory access on multiprocessor architectures and how scheduler handles them.

2.1 Completely Fair Scheduler

Completely Fair Scheduler (CFS) is the current Linux process scheduler, which was merged into the version 2.6.23 of the Linux kernel in 2007. Its author is Ingo Molnár, who is the author of the previous *O(1) scheduler* as well.

CFS features queuing tasks in a red-black tree structure ordered by time spent running on the CPU so far. Red-black tree is a binary search tree with self-balancing mechanism based on marking nodes with either red or black color. When the scheduler needs to choose the next task to run, it takes the leftmost node with the lowest execution time.

The time complexity of the CFS scheduling is O(log N), where N is the number of running tasks. Taking the leftmost node with the next scheduled task can be done in a constant time, but queuing the task requires O(log N) operations to insert it back into the red-black tree. Even with a higher scheduling complexity, the CFS scheduler has a better fairness and responsiveness than the previous O(1) scheduler, which used a simple queue to choose the next task.

On multi-core systems, the scheduler uses a separate queue for each core. In order to effectively use the processing power, the scheduler must regularly balance those queues by moving processes from the most busy cores to idle ones.

When moving processes between cores, scheduler takes in the account a topology of the system. Losing data from caches after the migration can have a bigger impact on performance than leaving the process on the busy core.

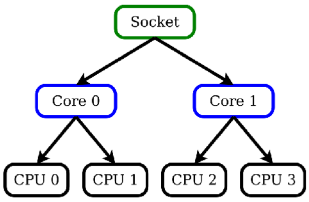


Figure 2.1: Hierarchy of scheduling domains[3]

CFS solves this problem by using scheduling domains [2]. Scheduling domain is a set of CPUs that should be balanced between themselves. CPUs in a scheduling domain are divided into groups. Scheduler then checks the load of the whole groups to decide if there is a need to migrate processes between them.

There are multiple levels of scheduling domains with different parameters such as how often is the load difference checked or how big the load difference between groups must be to migrate tasks to balance the queues. The lowest level is between hyper-threaded logical cores where there are almost no losses of cached data and rebalancing can be done quite often. A higher level is between physical processor cores where cache losses can have bigger impact on the decision to migrate the task. Above those cores can be processor sockets on machines with multiple physical processors with different access speed to different memory sections.

Scheduling domains are regularly rebalanced by going up from bottom of the scheduling domain hierarchy (illustrated by Figure 2.1) and by checking balance of the groups on each level.

2.2 Scheduling on SMP systems

Symmetric multiprocessing (SMP) is an architecture of computers with multiple physical processors that have a single shared memory, a shared access to all IO devices, and that run on the same instance of operating system. This allows the machine to offer more processing power with a little overhead caused by memory sharing.

Each processor has still its own high speed cache, but due to memory sharing, *cache coherence* must be maintained – the data shared between processors in their caches must be uniform.

There are two ways of accessing the shared memory from multiple processors: uniform (UMA) and non-uniform (NUMA) memory access. When correctly used, the NUMA technology has higher performance capability with similar configuration compared to UMA systems. In our experience most paying customers of Red Hat use the NUMA technology and we will primarily focus on scheduler behavior on this SMP architecture.

2.2.1 Uniform memory access

In the UMA architecture, all processors share a single memory controller that they use to access the shared memory. Therefore, each processor has the same speed of memory access and the same latency. They share a common access route to the memory, which brings more simplicity at the cost of a lower bandwidth and speed.

In this architecture it is easier for the scheduler to balance processes between the physical processors. The time to access the shared memory is the same on all of the cores and

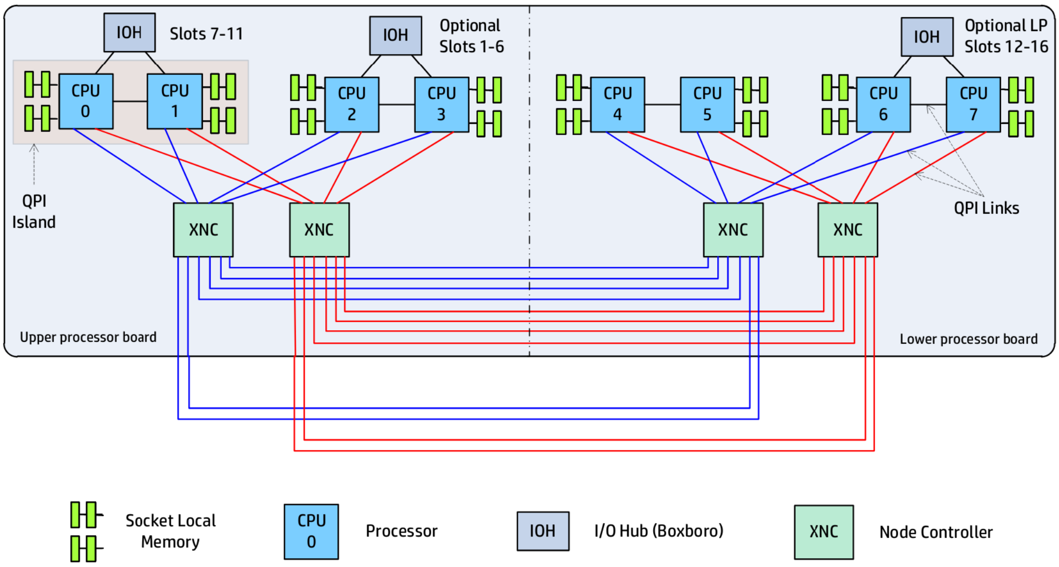


Figure 2.2: Architecture of NUMA communication on HP ProLiant DL980 [6]. Each processor connected with its own local memory makes up a NUMA node. Each pair of nodes have dedicated interconnect bus for faster data transfer between them. Communication with other nodes is realized through node controllers. The topology of the interconnect buses shows there will be 4 different access speeds depending on the distance between nodes. The local access is the fastest, then follow the access to the neighbor node and the access through one controller, and the slowest is the access through both controllers. The interconnect buses are doubled to avoid overloading one of them.

therefore there is no need to move the memory associated with a process to any other place for faster access.

2.2.2 Non-uniform memory access

The NUMA architecture tries to solve the problem with low bandwidth. It arranges physical processors or cores into nodes, where each node has its own separate memory and a bus for faster access. This significantly improves overall memory throughput of the system when used correctly.

Nodes also have to be connected to each other to access memory of other nodes. That is achieved using either interconnecting buses or controllers. Each manufacturer has its own technology implementing the interconnection: Intel uses Ultra Path Interconnect which replaced its QuickPath Interconnect from older machines. On the contrary, AMD uses Infinity Fabric supersetting the older HyperTransport.

On bigger machines with a large amount of processors, not every two processors are necessarily connected. Instead, they may access the data through a path of connected nodes. This can be seen in Figure 2.2 showing an 8 NUMA node machine with an advanced structure of interconnect buses and controllers.

Consequence of interconnection between NUMA nodes is the different latency between nodes which must be taken into account when balancing tasks between nodes. Difference in the access latency between a pair of NUMA nodes for the machine from Figure 2.2 can be seen in the following part of output from a command `numactl --hardware`:

```
node distances:
node 0 1 2 3 4 5 6 7
0: 10 12 17 17 19 19 19 19
1: 12 10 17 17 19 19 19 19
2: 17 17 10 12 19 19 19 19
3: 17 17 12 10 19 19 19 19
4: 19 19 19 19 10 12 17 17
5: 19 19 19 19 12 10 17 17
6: 19 19 19 19 17 17 10 12
7: 19 19 19 19 17 17 12 10
```

The `numactl` utility also provides possibility of pinning processes to a specific NUMA node to ensure the process and its memory will stay on the intended node. This allows user to arrange processes manually in a way considered as the best for maximum performance.

Balancing tasks between NUMA nodes is difficult for the scheduler since it needs to take into account an expensive memory movement or access to different nodes. With a wrong approach the performance of a NUMA system can drop even below the performance of a similar UMA system[1].

Balancing processes between NUMA nodes is still in active development, which brings many changes. These usually improve performance, however, there are cases when a change can cause a performance regression. Therefore, it is essential to carry out a thorough performance testing of the scheduling.

[1] http://highscalability.com/blog/2013/5/30/google-finds-numa-up-to-20-slower-for-gmail-and-websearch.html

Chapter 3

Performance measurement

Performance testing is examination of the system behavior under a workload and of its effectivity of resource usage. For many systems, the time they are able to respond in is crucial and this property affects the usability of the system.

Compared to functional testing, performance testing does not produce an exact true or false result. It produces a set of numerical values which must be compared to presumed values or values from an other version to make a conclusion.

After the performance measurement, the next step is inspection of behavior of the system to understand the measured values and to determine the causes of the difference from the expected values.

In order to measure the performance of a scheduler, we use benchmarks. Benchmarks generate artificial load imitating the load in real environment. While stress testing the system, they also measure its performance. The benchmarks typically return a value representing the performance of the system. The value is usually in the form of the time that the task needed to finish or of the amount of the operations that the system could perform per a unit of time.

In the previous chapter, we have introduced the current state of task scheduling in the Linux kernel. Effectivity of the scheduling and of task migration between processors affects the amount of the tasks that the system can handle and the time that a task spends before finishing.

Although the available benchmarks generate values that are suitable for comparison, they do not provide any more detailed information about how the system achieved the measured performance or where the possible bottlenecks could be [5].

To get a better insight into the behavior of the system, there are many tools to collect performance records about the system behavior. Useful information about the scheduler include assignment of tasks to the processor cores, the time that the tasks spent out of the CPU in queues, load of each processor core, or the location of memory of the processes on NUMA systems.

3.1 Performance metrics

We now present typical metrics used in performance measurement.

Throughput These metrics represents the amount of operations per time unit. The operations can be, e.g., floating point operations, or operation as specific tasks defined by a benchmark (e.g. SPEC Java benchmarks). Time units are usually seconds. Higher values mean better performance.

Run time This metric is simply the real time that the benchmark needed to finish the execution. It is mostly presented in seconds, but longer runs can be presented in a more human-friendly format, converting the time to hours, minutes, and seconds. The lower the run time is, the better performance it represents.

Time out of CPU This metric represents the time that the benchmark spent in process queue waiting for execution. This metric can be calculated from user, kernel, and real time provided by the *time* utility and from the number of threads, that the application used:

$$T_{out\_of\_cpu} = (T_{real} \times used\_threads) - (T_{user} + T_{kernel})$$

The lower this value is, the better behavior of the scheduler it represents.

3.2 Benchmarks

In the following section we will describe the benchmarks that are currently used to evaluate performance of the latest kernel versions. The benchmarks are usually based on real applications used both in scientific and in business environments and are meant to stress test the system.

The benchmarks run in many threads or processes and feature communication between them to utilize communication between processors. A bad distribution of processes and threads by the scheduler increases their time of waiting in the queue to rise and the performance of the system naturally goes down. Moreover, on NUMA systems, the performance depends also on placement of data in the memory.

3.2.1 NAS Parallel Benchmarks

NAS Parallel Benchmarks [1] is a set of programs focused on performance of highly parallel computations on supercomputers. In addition to floating point computations, it targets communication and data movement among computation nodes. The performed algorithms are based on large scale computational fluid dynamics at the Numerical Aerodynamic Simulation (NAS) Program which is based at NASA Ames Research Center.

Benchmarks are written in Fortran-90 or in C language, as these were the most commonly used programming languages in scientific parallel computing community at the time when the benchmarks were created. They can be compiled with different classes of problem sizes to suit machines with different amount of memory and of computational power.

The main output value of the benchmark is the throughput measured in units called Mop/s (millions of operations per second) representing the amount of floating-point operations per unit of time.

The benchmark also offers a few parameters that can be passed to the benchmark before the execution. One of them is the number of computation threads, which in a lower amount slows down the run time, but allows one to measure behavior of the system without full usage. Figure 3.1 shows an example of a throughput with different number of threads on a machine with 24 physical cores and hyper-threading.

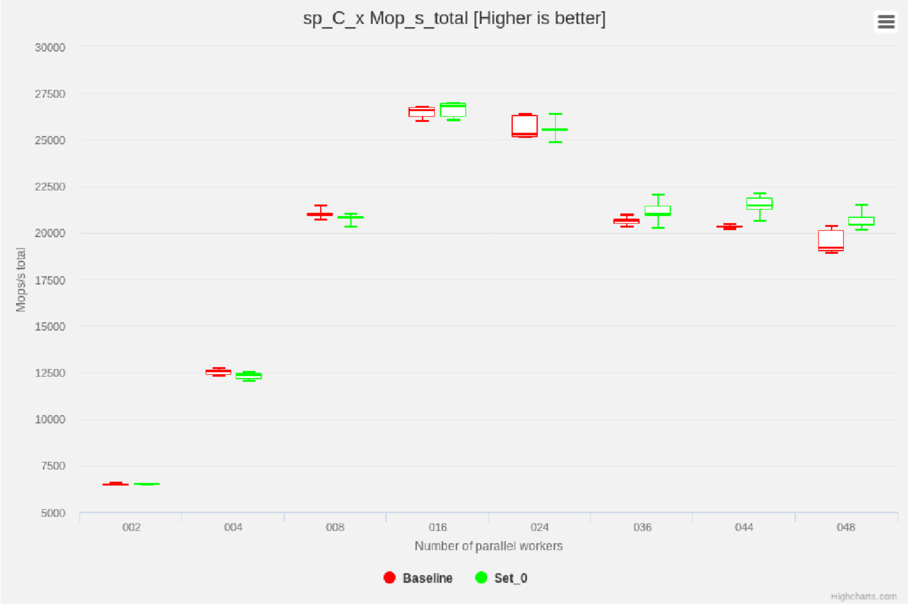


Figure 3.1: Example of Scalar Penta-diagonal solver results from the NAS Parallel benchmark with different number of computational threads. The size of the boxes represent the inaccuracy of measurement caused by noise and by non-deterministic behavior of scheduler.

The downside of this benchmark is it can only run with a fixed dataset, but not for a fixed time period. This constraint makes the run time of the benchmark with less threads longer.

3.2.2 SPECjbb2005

Java Business Benchmark [11] created by Standard Performance Evaluation Corporation (SPEC) behaves as a server-side Java application. It is primarily focused on measuring performance of Java core implementation, but it also reflects a performance of operating system and of the CPU itself. It models a system of a wholesale company as a multitier application. The benchmark itself creates the load, measures the throughput, and also generates a simple report in HTML and raw text formats.

The main output value is *throughput* measured in units called *SPECjbb2005 bops* [1]. In case we use more JVM[2] instances, there is a second unit called SPECjbb2005 bops/JVM representing an average throughput of a single JVM instance. The collected metric is memory consumption, which is not that important for scheduler performance monitoring.

3.2.3 SPECjvm2008

Java Virtual Machine Benchmark [12] is another benchmark from SPEC focused on Java Virtual Machine and Java Runtime Environment, but reflecting also behavior of process scheduler and memory management.

It consists of separate operations using real life applications (e.g. Sunflow rendering, Java compiler) or stressing specific part of Java implementation (e.g. cryptography algo-

[1] Business operations per second
[2] Java virtual machine

rithms, working with XML documents, Scimark floating point benchmark). The operations run in multiple threads sharing data both on application level and in common JVM instance.

Output of the benchmark uses *ops/m* unit describing number of executed operations per minute.

3.2.4 LINPACK benchmark

LINPACK Benchmark[4] comes from the LINPACK package, which was used to solve systems of linear equations. It is primarily used to measure floating point computation power of large machines at both single and double precision. It is used also by the TOP500 project for building list of 500 most powerful computers.

Main measured value from the benchmark is the number of floating point operations per second (*flops*), but the benchmark output provides more information such as run time, page faults, context switches, or location on which core or NUMA node it ran.

3.2.5 STREAM benchmark

The STREAM benchmark [9] is a simple benchmark aimed for measuring memory bandwidth and also computation speed of simple vector kernels. Its motivation is a slow grow of memory performance compared to CPU, which makes speed of this component also crucial for the performance of the whole system.

Output metric of the benchmark is memory bandwidth in *MB/s*, but it provides the same statistics as the Linpack benchmark: run time, page faults, context switches, or location on which core or NUMA node it ran.

3.3 Performance analysis tools

Although benchmarks create workload on tested system and collect some metric of its performance, tools for performance analysis are needed to get better insight to behavior of the system. To the interesting information belong utilization of processor cores, migration of benchmark process and its memory between NUMA nodes, or the time the process spent in queue out of CPU. This section describes utilities used to collect data to get the insight to system behavior.

3.3.1 time

Time is a simple command for measuring the time that an application spends running. The most common numbers it reports are the total real time that the application needed to finish, the time that it spent in the user mode, and the time spent in the kernel mode.

Many benchmarks provide the execution time themselves which can make this utility unnecessary. However, the interesting metric is the time that the application spent out of the CPU waiting in queue. This value is not provided directly, but time provides user, kernel and real time from which the metric can be calculated as described in Section 3.1.

It can be confused with the Bash builtin command `time`, which provides similar output, but the real binary can provide more verbose information with the possibility of custom formatting of output. It can be usually called from `/usr/bin/time`.

3.3.2 ps

Ps is a Linux command, which is used to display information about active processes. Its name stands for “processes status”. It can provide various information obtainable from the virtual files in the `/proc` directory. The most common information include the process PID, time spent on processor, state of the process, used memory, associated terminal, the command that started the process, and more.

Especially useful are the optional columns `PSR` and `NUMA`. They show the number of the processor and the number of the NUMA node where the process is running. Continuous monitoring of those values can provide view on migration of the process during its run time.

Output of the command can be filtered in many ways. By default, it shows only processes for the current user and for the current terminal, but it can list all the processes on the system. The listing can be filtered using parameters by most of the columns of information it provides. The listing can be limited for instance by a specific terminal, by an effective user, by children of a specified process, or by a PID to a single intended process.

3.3.3 mpstat

This Linux command provides continuous information about utilization of CPUs. It can show utilization of processor cores, of NUMA nodes, or of the whole system dependent on passed argument. Some of the provided values are utilization in user space, utilization in kernel space, percentage of waiting for IO operations, percentage of handling interrupt requests and the idle percentage, when system is idle and does not wait for any IO operation.

Mpstat can collect those data once when executed or at regular time intervals. The regular collecting of utilization of the CPU cores or of the NUMA nodes is done through the run time of a benchmark. With little processing of the data, it is easy to watch whether the distribution of load between the cores and the NUMA nodes is equal or not.

3.3.4 turbostat

This tool provides measuring of hardware properties of the CPUs with the x86 architecture. It reports for each core its usage, frequency, temperature, and percentage of time in different idle states. For each socket it reports its power consumption.

There are two ways to run turbostat. It can be supplied with a command to run and it will return the average values from the run time of the command. Without the command it will collect the statistics at regular time intervals.

Data from this tool can be used to analyze performance drop caused by the CPU itself. This can happen due to frequency drop because of overheating or of missing workload. The power consumption data can be used to roughly compare the power efficiency of both the scheduler and the physical CPUs, but the command only provides consumption of the CPUs and their RAM and not of the whole machine.

3.3.5 perf

Linux command `perf`, also known as `perf_events`, is a tool for profiling with performance counters Linux subsystem. It provides counting of *hardware events* (e.g. cpu cycles, branch and cache misses), *software events* (e.g. context switches, page faults), or custom *tracepoints* (e.g. specific system calls, filesystem or network operations).

It offers wide range of commands, from which the most used are:

`perf stat` The command counts selected events during an execution of a process or during a specified time period. It can observe events belonging to the process or system-wide. The counted statistics are written at the end of the time interval or of the process execution.

`perf record` The command record the events to perf.data file for later analysis.

`perf report` The command reads the perf.data file created by perf record and displays the collected statistics.

`perf top` The command provides live analysis of system by showing all observed function calls ordered by the number of cycles spent in them.

Chapter 4

Processing of results

Getting the output of benchmarks and tools for performance analysis is just a part of performance regression testing. To perform comparison of two results, it is essential to store the results in an efficient way for quick creation of a comparison report. The quality of the comparison report also affects the right choice and a use of visualization of the results and their comparison.

This chapter is structured as follows. Section 4.1 describes ways of storing the results from benchmarks and Section 4.2 describes different visualization methods suitable for analysis of performance changes.

4.1 Storage of the results

Benchmarks sometimes generate long human-readable output in text or even HTML format. This is useful when analyzing a single report. In the output there are details of the test run itself, a simple resource usage, or success of result validation. However, the amount of result starts to rise with repeated runs, with different amount of instances, and with new versions kernels.

For the comparison of performance results, the number representing throughput or time of each benchmark run is usually enough. Those numbers can be preprocessed from the benchmark output files to a format more suitable for quick accessing of the required data.

4.1.1 XML files

XML is a markup language, that can store heterogeneous data in a tree structure. The tree structure can effectively represent the test scenario running each benchmark operation with different parameters and multiple repetitions.

Another feature of XML format is human-readability offering quick insight to stored data just with any text editor. This comes with a disadvantage of redundant data in the form of repeated names of tags and attributes which often take more space than the data itself. Parsing of the data also takes considerable amount of the CPU time prolonging the duration of analysis.

In our team, we use the XML format for storing result values from benchmark runs and their aggregated statistics for easier generation of performance comparison reports. Next to the results is also stored configuration of the benchmark run containing properties of the system that the benchmark ran on. The properties include hostname of the testing machine, version of kernel and of operating system, or configuration of the environment.

```
<?xml version="1.0"?>
<beaker_run_result>
  <test_result>
    <nas_result benchmark_name="mg_C_x">
      <threads number="2">
        <result mops="5560.4" real_time="31.0" out_of_cpu_time="0.9"/>
        <result mops="5411.4" real_time="32.2" out_of_cpu_time="1.6"/>
        <result mops="5499.3" real_time="31.4" out_of_cpu_time="0.6"/>
        <result mops="5407.4" real_time="31.9" out_of_cpu_time="1.3"/>
        <result mops="5376.1" real_time="32.0" out_of_cpu_time="0.7"/>
      </threads>
      <threads number="4">
        <result mops="10254.9" real_time="16.8" out_of_cpu_time="1.2"/>
        <result mops="10075.4" real_time="17.1" out_of_cpu_time="1.7"/>
        <result mops="10226.6" real_time="16.8" out_of_cpu_time="1.4"/>
        <result mops="10250.2" real_time="16.8" out_of_cpu_time="1.3"/>
        <result mops="10227.7" real_time="17.0" out_of_cpu_time="2.5"/>
      </threads>
...
```

Figure 4.1: This example shows beginning of XML file with important values from one NAS Parallel benchmark run scenario. The XML file starts with root element `<beaker_run_result>` and `<test_run>` node which are wrapping `<nas_result>` nodes representing results from each benchmark operation from NAS Parallel benchmark suite. Each benchmark operation is run with different amount of threads in few loops to lower the measurement inaccuracy. Nodes of results with the same number of threads are wrapped in `<threads>` node. All the values are stored as attributes of the corresponding node.

Example of results from one run of the NAS Parallel benchmark scenario stored in the XML format is in Figure 4.1. Example with aggregated data is in Figure 4.2. Example of an XML file with properties of benchmark run result is shown in Figure 4.3.

4.1.2 Relational database

Relational database is a type of database using relational model. The relational model stores data in tables using rows for records and columns for their attributes. Each row represents a unique record with its attributes in columns. Columns store values of attributes with the same data type. Records in different tables can be connected in relationships.

There are many database management systems implementing the relational database model available under various licenses. From the open-source we can name PostgreSQL, SQLite, MySQL, or its fork MariaDB. To the category with proprietary code belong implementations from companies including Oracle, Microsoft, or IBM.

Data in database are managed using SQL (Structured Query Language). It provides commands for storing, manipulating, and retrieving data. With advanced joining of tables and filtering it provides wide possibilities of data processing just at the point of reading of the stored data.

Database offers much faster access to data without complicated parsing of text files. Searching through the data can be much faster with indexing of the records by selected

```
<?xml version="1.0"?>
<beaker_run_result>
  <test_result>
    <nas_result benchmark_name="mg_C_x">
      <threads number="2">
        <mops mean="5450.9" stdev="68.4" first_q="5407.4"
          median="5411.4" third_q="5499.3" max="5560.4" min="5376.1"/>
        <total_time mean="31.7" stdev="0.4" first_q="31.4"
          median="31.9" third_q="32.0" max="32.2" min="31.0"/>
        <out_of_cpu_time mean="1.0" stdev="0.4" first_q="0.7"
          median="0.9" third_q="1.3" max="1.6" min="0.6"/>
      </threads>
      <threads number="4">
        <mops mean="10207.0" stdev="66.8" first_q="10226.6"
          median="10227.7" third_q="10250.2" max="10254.9" min="10075.4"/>
        <total_time mean="16.9" stdev="0.1" first_q="16.8"
          median="16.8" third_q="17.0" max="17.1" min="16.8"/>
        <out_of_cpu_time mean="1.6" stdev="0.5" first_q="1.3"
          median="1.4" third_q="1.7" max="2.5" min="1.2"/>
      </threads>
...
```

Figure 4.2: Another form of stored data shows this beginning of XML file. Instead of all values obtained from the benchmark run scenario, here are only aggregated statistical values form the sets of collected values from each configuration. The aggregated values include minimum, maximum, mean, median, quartiles and standard deviation of metrics like throughput, run time, or time in queue for CPU. Those values are directly usable for plotting of comparison graph without any manipulation with them lowering the time for generation of reports.

```
<?xml version="1.0"?>
<beaker_run_result>
  <settings>
    <BenchmarkName value="NASParallel"/>
    <Distribution value="RHEL-7.5"/>
    <Kernel value="kernel-3.10.0-862.el7.x86_64"/>
    <Architecture value="x86_64"/>
    <TunedProfile value="throughput-performance"/>
    <SELinux value="Enforcing"/>
...
```

Figure 4.3: Example of XML file with properties of a benchmark run. In a flat structure in node `<settings>` are key-value pairs stored as node name and attribute `value` with the actual value of the property. There are information including benchmark name, version of kernel and operating system, hostname and architecture of testing machine and various system and environment parameters that could affect the performance measurement results.

columns. Moreover, databases store data more space-efficiently directly in binary format to avoid unnecessary conversions.

The efficiency of database comes with its disadvantages. Storing the data in binary form eliminates the possibility of quick insight to data like with XML files. To access any data, it is required to write an SQL query to request specific information from the storage. More complications come with design of the database tables. Different benchmarks produce different type of result and the requirements for the stored data can change over time. This requires building universal complex structures or occasional changes in the database model.

In our team we currently consider the option of storing results of benchmarks in a database, which would require a lot of work needed for migration from the current storing in XML files.

4.2 Visualization the results

Effective analysis of results from a performance measurement requires delivering the comparison in a form in which a human can quickly see the differences in measured values and their severity.

Visualization offers this advantage against raw text data collected from benchmarks and performance analysis tools. It allows us to much faster see important relations between the collected data utilizing often smaller display area than the raw data.

Right visualization also allows us to deliver more easily understandable data even for people that do not work with performance analysis on their daily basis.

4.2.1 Line graphs

Line graphs are the simplest method of displaying a course of values of a variable dependent on a parameter in two dimensional space. It allows to easily spot nature of the plotted values – either increasing, decreasing, or constant. The data can be plotted as discrete values using points or as a continuous function using a line. Comparison of more variables from different datasets is done by plotting multiple lines to the same graph, one for each variable. In Figure 4.4 is an example of line graphs showing CPU utilization of system NUMA nodes and the ratio of access to memory of remote nodes.

Although line graphs are quick and simple to create and use, they fail to scale for larger amount of lines in a single graph. Larger amount of lines becomes too confusing and impossible to read. Graphs of CPU usage of each NUMA node in Figure 4.4 is still readable, but impossible to use for utilization of every CPU core.

4.2.2 Heat maps

Heat maps are three dimensional graphs which are using color as the third dimension for values. This allows to plot two dependencies of the values compared to line graphs, which must use multiple lines to plot the same data. Heat maps provide better scalability for larger data, where line graphs would be confusing with too many lines. It is also much easier to spot correlation with the additional dimensions compared to line graphs.

In Figure 4.5 is a heat map showing utilization of all CPU cores over time under workload. The data was collected by the mpstat utility and processed to show the sum of user and kernel space utilization of each core. Plotting those data using line graph with a line for every core would be confusing even for this relatively small amount of CPUs.

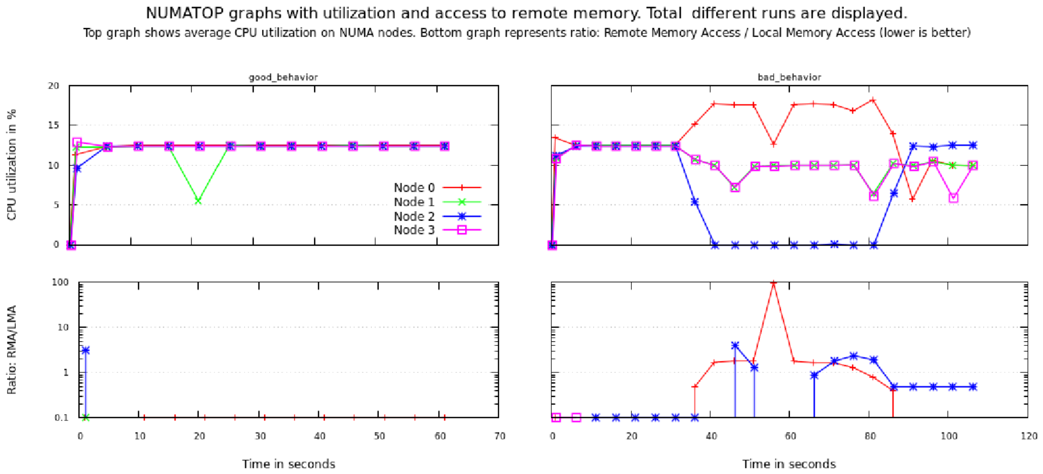


Figure 4.4: An example of line graphs showing statistics collected using the numatop utility. The graphs in the top row show CPU utilization of each NUMA node through time and in the graphs bottom row show ratio of access to local memory of the node and to memory of remote nodes. The graphs on the left show an expected behavior of a scheduler causing uniform utilization of each node and a minimal amount of access to memory of remote nodes. The utilization graphs use linear y axis and memory access graphs use logarithmic y axis. The plotted values were collected on 4 NUMA node machine under workload from the NAS Parallel benchmark running in 4 threads.

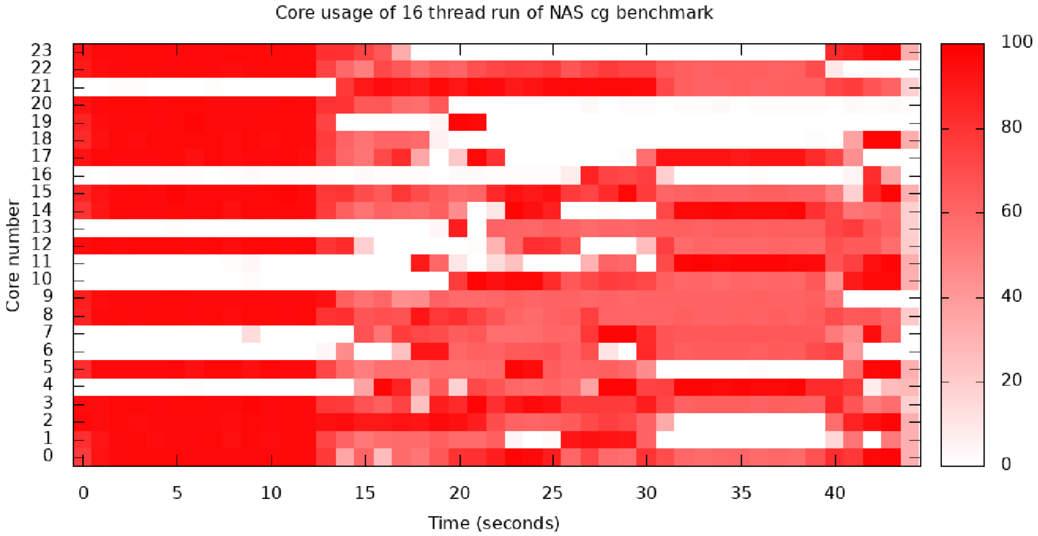


Figure 4.5: Example of a heat map showing CPU utilization over time. The machine with 24 logical CPUs is under a workload from NAS Parallel benchmark running in 16 threads.

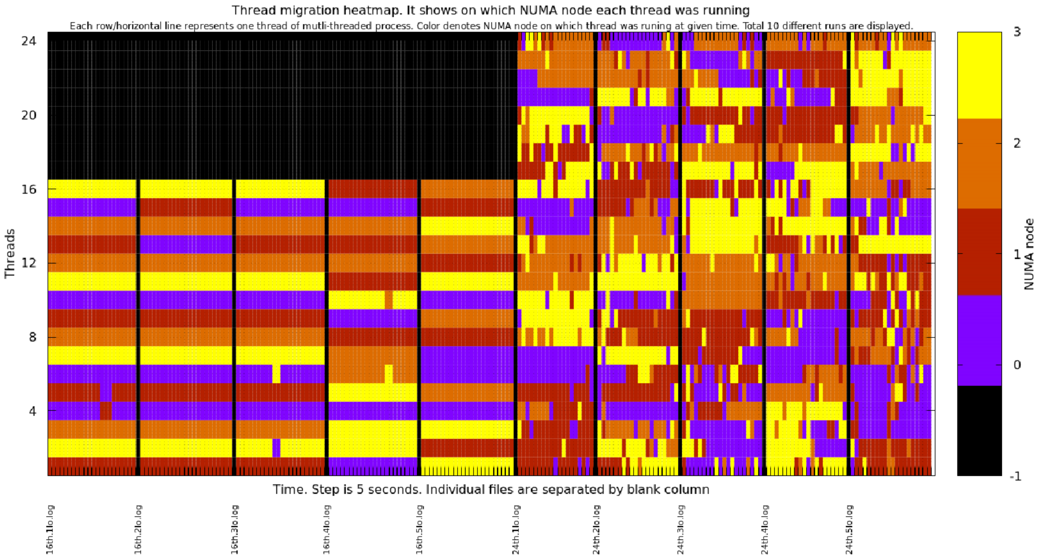


Figure 4.6: Example of a heat map showing thread migration between NUMA nodes. The expected result is not the highest value, but the minimum of color changes in each line. The shown result comes from a machine with 24 logical CPUs running the NAS Parallel benchmark on 16 and 24 threads in 5 loops.

Another use of heat map is shown in Figure 4.6. It does not show utilization of threads, but their location on which NUMA node collected by ps utility. This heat map shows the migration of threads between NUMA nodes and the expected result is not the highest value, but minimum of color changes in each line. The shown data comes from NAS Parallel benchmark, which was run with 16 and 24 threads in 5 loops. The heat map shows better scheduler behavior on the 16 threads run than on the 24 threads run.

4.2.3 Box plots

Box plot is a method for displaying statistical properties of data from multiple measurements. It extends the simple visualization of discrete values by adding to the median values also the minimum and the maximum measured values and the first and the third quartiles from the measurement.

In some cases the whiskers for minimum and maximum value represent standard deviation or the 2$^{\text{nd}}$ and the 99$^{\text{th}}$ percentile. Data out of the range is displayed as standalone points above or below the whiskers.

Box plots are great to illustrate dispersion and variation of the data that do not follow exact normal distribution. All of the displayed marks show accuracy and reliability of measurement. This insight helps to distinguish real performance regression from a noise caused by unpredictable behavior of the scheduler and measurement error.

In Figure 4.7 is an example of a box plot showing throughput measured by the NAS benchmark with different number of threads.

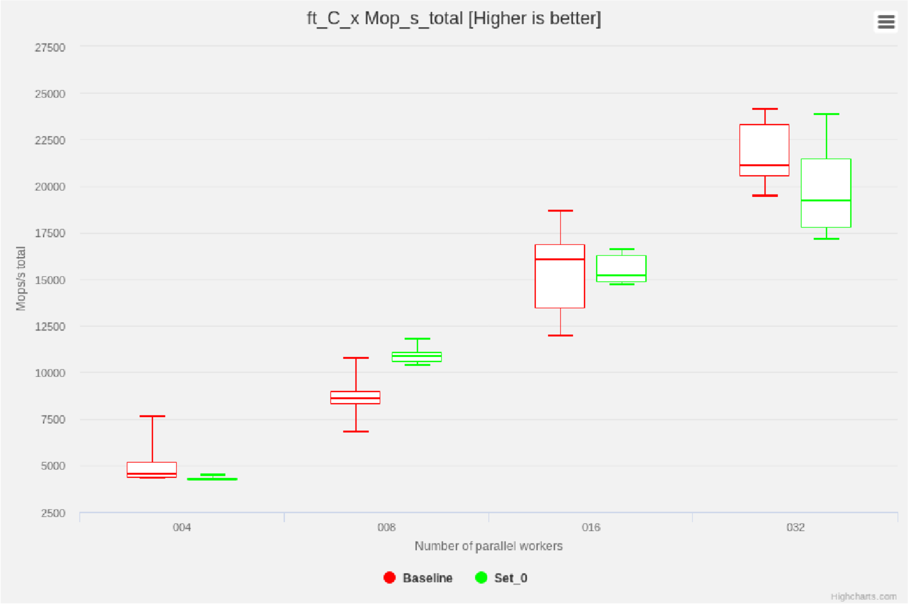


Figure 4.7: Example of a box plot showing throughput measured by the NAS benchmark from multiple measurements with different number of threads. The last two boxes and their whiskers show us that the range of measured values is the same despite the big difference in medians. Therefore the plotted result is treated as passed (without any performance regression).

Chapter 5

Visualization using timelines

A common way to analyze performance reports is to compare two results measured for different versions or settings. Usually, these are called the baseline and the target results or profiles. The comparison of two results allows one to interpret the measurement and changes between versions, usually containing clues to the cause of possible performance changes, e.g. change in some value of parameter or new code functionality.

However, sometimes it is not enough to compare just two strictly following versions. If, instead, we analyze a larger amount of results over a longer period of time, we can begin to see a whole new perspective. In that case there can be a much more visible difference between a deviation from a measurement error and a performance change. It is also easier to find versions where a performance degradation occurred and where it was fixed.

With larger amount of data, we can also detect creeping performance drops, which appeared continuously over a longer period of time and could not be detected, because they were within the tolerance due to the measurement deviation.

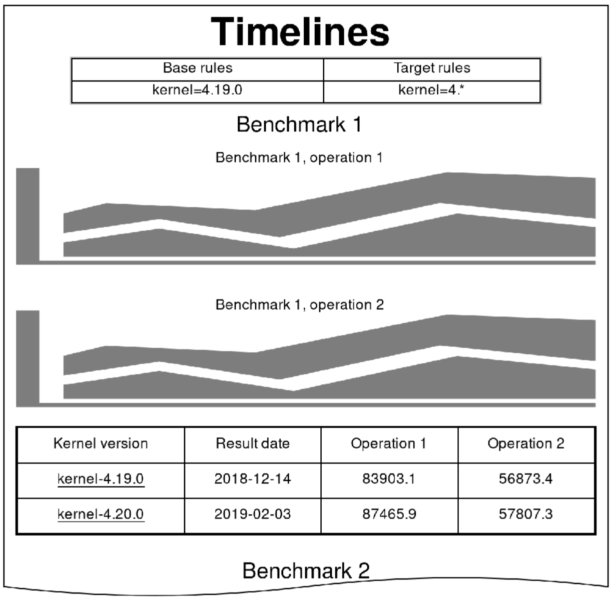
This chapter proposes new kind of reports with comparison of multiple results of performance measurement from benchmarks used by Red Hat Kernel Performance Quality Engineering team. The visualization of performance results helps to see the performance of Linux kernel scheduler in wider range of time as well as to determine stability and precision of different benchmarks.

5.1 Design

We mainly focus on performance of kernel versions on a specific testing machine. The timelines generator will output HTML reports showing benchmark results of the desired kernel versions on a single machine. We describe the mockup of produced HTML page in Figure 5.1. Displayed results are specified using rules for a base kernel as the first reference result and target kernels forming the actual timeline.

We propose that the most suitable type of graph for displaying results with repeated runs are boxplots. Boxplots show important statistical values from the runs: the median, the minimum, the maximum, and the first and the third quartiles. Those values can quickly reveal stability of the plotted benchmark and noise in the measurement that can help to distinguish true performance degradations.

Each graph of benchmark operation contains results from all thread configurations, but only one desired configuration is visible by default. The desired number of threads is the



Timelines

Base rules	Target rules
kernel=4.19.0	kernel=4.*

Benchmark 1

Benchmark 1, operation 1

Benchmark 1, operation 2

Kernel version	Result date	Operation 1	Operation 2
kernel-4.19.0	2018-12-14	83903.1	56873.4
kernel-4.20.0	2019-02-03	87465.9	57807.3

Benchmark 2

Figure 5.1: Mockup of timelines report page. The report contains results from a single testing machine fulfilling the specified rules. Below the table we display with the rules we display graphs of the available benchmarks. Each benchmark section will have a separate graph for each of its operations and a table below the graphs containing featured values from each operation for each benchmark with links to the results in our result database.

point where performance regressions create the biggest difference, which is in most cases the highest number of threads.

The graphs will also contain horizontal lines in background following median of the base result and its value increased and decreased by 5%. Those lines will allow more effective recognition of significant performance changes without looking at the absolute values of the measurements.

Under the graphs of all operations of benchmark will be a table containing medians of featured thread runs for each benchmark operation of every displayed kernel for browsing of the absolute result values. Each record will also work as a link to result record in the database of benchmark results of Red Hat Kernel Performance QE team.

5.2 Implementation

This section describes selected aspects of implementation of the timelines report generator. The generator is implemented in Python2 due to earlier origins of its implementation.

5.2.1 Comparison rules

For automatic report generation it is essential to allow defining rules, which will specify results, that can be used and in which role (i.e. whether they correspond baseline or target). We propose to use *regular expressions* to match properties of results. Regular expressions offer broad possibilities to describe shape of kernel version or the value of any environment configuration. E.g. to filter all builds of kernel 4.18 we can use simple pattern `kernel-4.18\..*`.

We store the rules in an XML file with the same node naming as in the XML file with the result properties. The first level of XML document contains three nodes representing the purpose of the rules.

- **Baseline rules** specify the first result in the plotted set. These act as the main result that the others are compared to. In case of multiple results fitting the rule, the newest one will be used.
- **Target rules** define the results to be plotted.
- **Starting rules** (optional) are for the case, when base result is not from the set of target results and there is need of specifying the first target result would be hard with regex.

5.2.2 Reading of results

Benchmark results are in our case stored in the filesystem in directories. The generator has to go recursively through directory with results of benchmark runs with files containing desired data. From each result, it starts with file containing properties of given result. This file provides metadata from the benchmark run including the time, machine hostname, kernel and OS version, benchmark name, configuration of environment for selecting desired results using the comparison rules. Example of the XML file with properties is in Figure 4.3.

After applying the rules the generator reads files from selected results with preprocessed data that are ready to use for drawing box plot graphs. Example of the file with preprocessed values is in Figure 4.2. Parsed data from this XML file is all the generator needs to start drawing the timeline graphs.

5.2.3 Plotting of timelines

For displaying the data in graphs using box plots we use Highcharts JavaScript library [7] for generating interactive SVG-based graphs for HTML pages. It offers free license for non-commercial use with available source code, but also paid licenses for commercial use. It provides fast and easy creation of various types of graphs including box plots with wide possibilities of customization.

On the HTML page are the charts rendered by JavaScript function from the Highcharts library to specified `div` element by its id attribute as an SVG image. The rendering function takes parameters for the graph in JSON format including type of chart, axis parameters, horizontal lines for reference median, interactive tooltips, and the data series.

5.2.4 Generator parallelization

With multiple entered sets of rules for timeline reports the generator can create multiple reports, e.g. separate reports for Red Hat Enterprise Linux 7 and 8. The procedure of parsing results and generating graphs is more on CPU than IO operations.

To reduce time needed to generate all reports multiprocessing is used. There is no need of exclusive access to any shared resource, because all of them are used only for reading, not writing. For implementation Python2 provides `multiprocessing` library with `Process` class that creates child process by forking allowing true parallel generation of reports. Using only threads in Python does not lead to parallel computation because of Python's *global interpreter lock* is preventing multiple threads from executing code simultaneously.

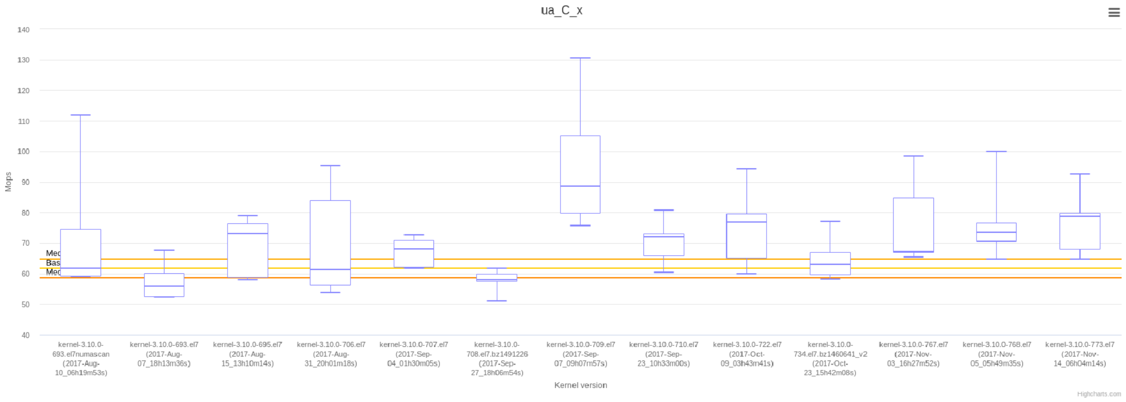


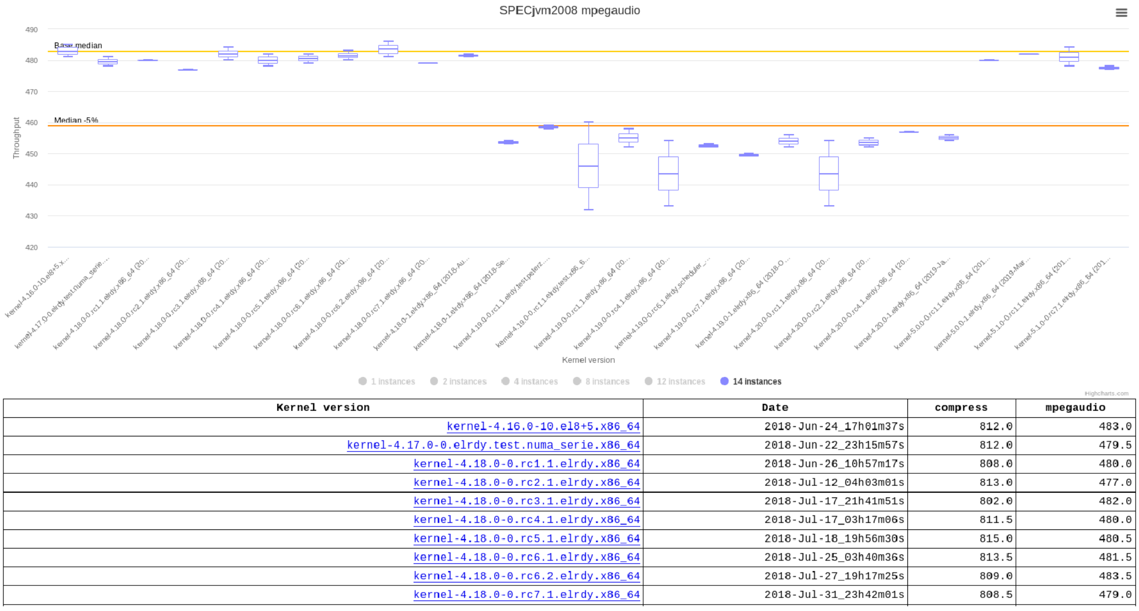
Figure 5.2: Example of very unstable benchmark which was removed from the run scenario because of irrelevant results. The yellow and orange lines show median of base result and 5% differences which are usually threshold for suspected performance regression.

5.3 Generated output

Output of the timeline generator is HTML report containing graphs of different benchmark operations with summary tables as described in mockup in Figure 5.1.

The graphs use box plot method which shows the measurement accuracy. This feature helped to eliminate unstable benchmark operations from run scenarios. One of the unstable operation is shown in Figure 5.2.

Thanks to wide range of displayed results it is easy to find version where a performance regression occurred, or was fixed. This case is captured in Figure 5.3, where is also shown part of the summary table. The cause of the degradation is change in memory migration rules between NUMA nodes. However, this change was not pure regression, because other benchmark shown significant gain displayed in Figure 5.4.



Kernel version	Date	compress	mpegaudio
kernel-4.16.0-10.el8+5.x86_64	2018-Jun-24_17h01m37s	812.0	483.0
kernel-4.17.0-0.elrdy.test.numa_serie.x86_64	2018-Jun-22_23h15m57s	812.0	479.5
kernel-4.18.0-0.rc1.1.elrdy.x86_64	2018-Jun-26_10h57m17s	808.0	480.0
kernel-4.18.0-0.rc2.1.elrdy.x86_64	2018-Jul-12_04h03m01s	813.0	477.0
kernel-4.18.0-0.rc3.1.elrdy.x86_64	2018-Jul-17_21h41m51s	802.0	482.0
kernel-4.18.0-0.rc4.1.elrdy.x86_64	2018-Jul-17_03h17m06s	811.5	480.0
kernel-4.18.0-0.rc5.1.elrdy.x86_64	2018-Jul-18_19h56m30s	815.0	480.5
kernel-4.18.0-0.rc6.1.elrdy.x86_64	2018-Jul-25_03h40m36s	813.5	481.5
kernel-4.18.0-0.rc6.2.elrdy.x86_64	2018-Jul-27_19h17m25s	809.0	483.5
kernel-4.18.0-0.rc7.1.elrdy.x86_64	2018-Jul-31_23h42m01s	808.5	479.0

Figure 5.3: Generated box plot timeline graph of *mpegaudio* operation from SPECjvm2008 benchmark with part of table showing numeric results of the SPECjvm2008 operations. The table shows medians from benchmark runs with featured number of threads and the kernel names are links to our team database of results. It displays performance results of upstream kernels for Red Hat Enterprise Linux 8. The performance degradation is consequence of changes in memory placement and migration behavior.

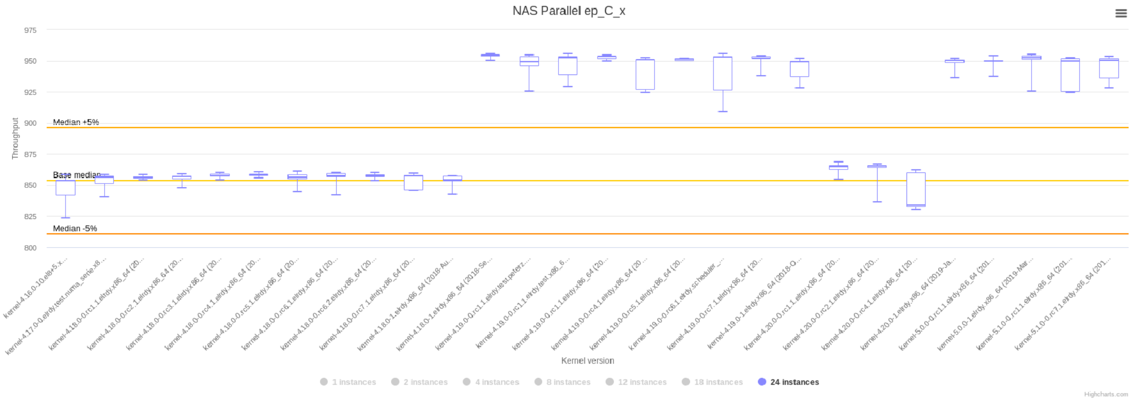


Figure 5.4: Timeline box plot from the same report as Figure 5.3 showing results of *Embarrassingly Parallel* operation from NAS Parallel benchmark. This benchmark got performance gain in the same version as the SPECjvm2008 benchmark registered performance degradation. Fortunately, this performance gain in NAS Parallel benchmark operation did not disappeared after fixing the degradation at SPECjvm2008 benchmark.

Chapter 6

Automatic evaluation

With every new release of regular kernel and its testing the amount of produced results rises. Number of results can rise with every new kernel version even by hundreds with more testing machines, with different configurations, with benchmarks with different focus, or with baselines from different supported versions.

In this thesis we attempt to automate the repetitive classification and labeling of results as passed (without any performance regression) or failed (containing performance regression). This allows us to skip the results without any significant change. Instead, more time is left for focusing on the results with performance regressions.

In this chapter we will describe the way of obtaining data from results and their processing for the evaluation. We will describe different classification models suitable for given dataset and compare their success rate.

6.1 Human classification of results

There are quite many results to check with every new tested kernel. The test suite with the Linpack, Stream, NAS Parellel and SPEC benchmarks are run on more than eight machines with different amount of NUMA nodes and processor models. Quality engineer has to go through the reports and check the results of all the benchmarks with different configurations manually.

When looking at results of benchmark operation, the most important values are medians from the runs with the different amount of threads. Unfortunately, due to the noise in measurement and limited amount of repeated runs the medians can be significantly affected by the noise. This complication brings the need of more detailed inspection of results than just checking the difference in medians of results.

Another useful metrics are the minimum, the maximum and the quartile values from each measurement. They reveal the stability of the benchmark and the measurement noise. With those values it is much easier and more accurate to tell the measurement noise from performance regression. With similar minimum, maximum and quartile values the performance can be the same even with the difference in median values.

The threshold between noise and performance regression is considered as 5% difference between base and target measurement but varies by the stability of each benchmark and complexity of the machine it was run on.

```
<comparison_results base_uuid="..." report_uuid="..." target_uuid="...">
  <comparison_result benchmark_name="NASParallel"
      operation_name="bt_C_x" result_status="pass">
    <result median_diff="1" threads_no="4">
      <base_result first_q="7409.2" max="7507.0" mean="7451.7"
        median="7456.5" min="7406.4" stdev="39.2" third_q="7479.3" />
      <target_result first_q="7503.0" max="7597.5" mean="7513.1"
        median="7522.2" min="7404.8" stdev="62.7" third_q="7537.8" />
    </result>
    <result median_diff="0" threads_no="8">
      <base_result first_q="14330.8" max="14740.1" mean="14476.6"
        median="14469.1" min="14325.9" stdev="151.7" third_q="14517.2" />
      <target_result first_q="14494.5" max="14619.3" mean="14481.1"
        median="14521.6" min="14233.7" stdev="130.5" third_q="14536.5" />
    </result>
...
```

Figure 6.1: Part of XML file with labeled data for classification.

6.2 Used technologies

For classification we use the *scikit-learn* library [10] with Python3. Scikit-learn is a library for data analysis and machine learning distributed under BSD open source license. It is easy to use with fast learning curve and well documented. It is a good choice for small and medium sized projects that do not need massive scalability. It provides various algorithms for classification, regression and clustering built on *NumPy* and *SciPy* Python libraries.

Unlike other machine learning libraries like PyTorch and TensorFlow, the scikit-learn library does not focus on deep neural networks for larger and advanced problems. It provides more simple classifiers for easier problems with smaller datasets where advanced methods would not have enough training data.

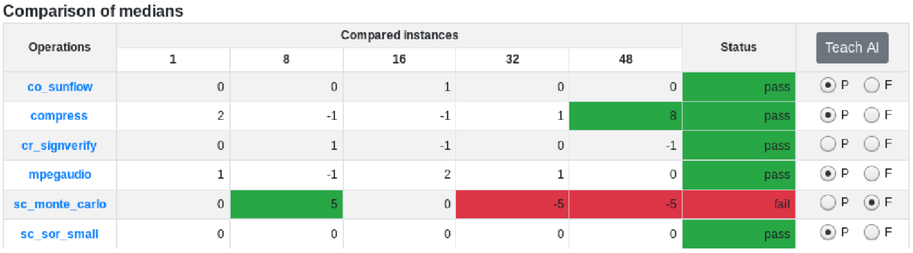
6.3 Reading the labeled results

Data used for learning are passed as an XML file containing preprocessed data from the base and the target run of the benchmark which are labeled as passed or failed.

Record of each benchmark operation is labeled as pass or fail and contains records of runs with different amount of threads or processes. Those records contain median, minimum, maximum and quartiles form the repeated runs of the configuration. The root element then contains UUIDs of the comparison and its base and target results for easier tracing in case of suspicious values. The example of XML file with data for the teaching of classifiers is shown in Figure 6.1.

6.4 Labeling of results

To teach the automatic classifier we need large amount of data for training. To reduce the time spent on labeling of the passed and failed results we included a HTML form to report page with comparison of two results to speed up this process. The design of the form is shown in Figure 6.2.



Comparison of medians

Operations	Compared instances					Status	Teach AI
	1	8	16	32	48		
co_sunflow	0	0	1	0	0	pass	● P ○ F
compress	2	-1	-1	1	8	pass	● P ○ F
cr_signverify	0	1	-1	0	-1	pass	○ P ○ F
mpegaudio	1	-1	2	1	0	pass	● P ○ F
sc_monte_carlo	0	5	0	-5	-5	fail	○ P ● F
sc_sor_small	0	0	0	0	0	pass	● P ○ F

Figure 6.2: Form from HTML comparison report page of SPECjvm2008 benchmark to label data for machine learning. The last column contains form manual labeling of the results which are sent using the button on the top. The *Status* column next to it displays the actual labels stored in the XML file with data for classification.

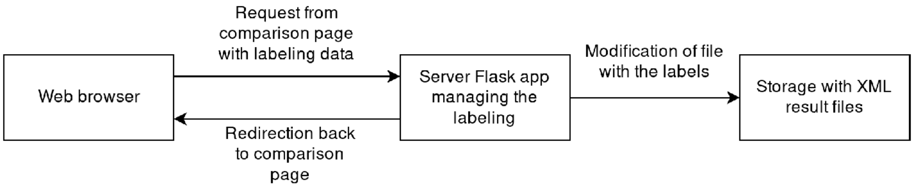

Web browser
Request from comparison page with labeling data
Redirection back to comparison page
Server Flask app managing the labeling
Modification of file with the labels
Storage with XML result files

Figure 6.3: Form from HTML comparison report page to label data for machine learning

For processing of the requests from this HTML form we built a simple HTTP simple server application using Flask framework for Python3. It is wrapped in the Docker container which runs on machine which stores the performance results and comparison reports. It receives a HTTP request with POST data containing the type of benchmark, path to directory with the report page and labels for results of each benchmark operation. Using this data the application modifies the XML file with labels described in Section 6.3 and returns HTTP redirect header back to the comparison report. This process is shown in diagram in Figure 6.3.

6.5 Preprocessing of the data for learning

We want to classify results of each benchmark operation with different thread configurations which is represented by `<comparison_result>` node in the example in Figure 6.1. The number of threads configurations represented by `threads_no` attribute in each `<result>` is not always the same and varies on the number of CPU cores on the testing machine.

The important values that we focus on are the statistical data from runs with different amount of threads, that one uses to label the data manually. From each of the runs we take minimum, median, maximum and first and third quartile from the repeated measurements of the same run configuration.

To avoid absolute values from measurements that are different on each machine, we use relative proportions. Difference of target and base medians are divided by base median and the rest of target statistical values are divided by the target median.

Next task is to reduce the provided data with variable number of threaded results to fixed size vector. We will take minimum, median and maximum from values of each statistical property: of minimums, medians, maximums and first and third quartiles. Each reduction will keep important values form the benchmark operation results in reasonably small vector.

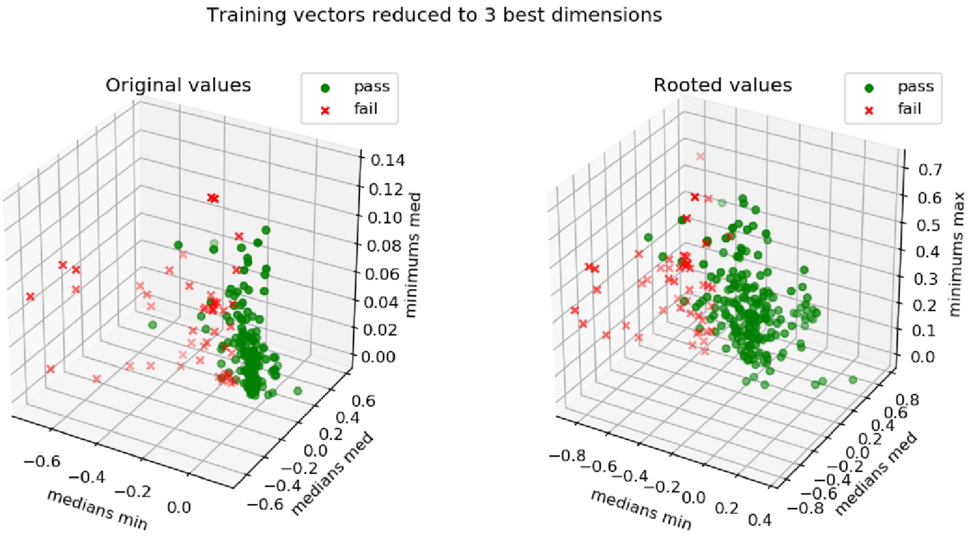


Figure 6.4: Plotted vectors from the training dataset reduced to 3 dimensions, which are the best for classification. The reduction was done by `SelectKBest` class from scikit-learn library by choosing the dimensions with the biggest variability between classes. The plot on the left shows scatter of vectors from dataset with unmodified values. The plot on the right shows the vectors from dataset modified by applying square root on their values. This modification and naming of the axis is explained in Section 6.5

The last operation with these vector values is applying square root. This operation reduces distances of vectors with exceptionally high values, which helps the k-NN and linear classifier to get better results. Scatter of the vectors plotted in 3 best dimensions is shown in Figure 6.4.

For later use, we will name the vector component like `medians min` (minimum value of all medians in results with different number of threads). The whole preprocessing of a single result of benchmark operation shows the Figure 6.5 with Python code snippet.

6.6 Comparison of classifiers

The classification is a procedure of assigning category to new observation based on training set of observations with specified category. Because of the availability of already classified data it belongs to *supervised learning* part of machine learning.

Data for classification are represented by set of vectors with fixed element count. Vector is an ordered set of numbers where each one represents a value in separate dimension of the source data. For supervised learning vectors from training set have assigned categories (also called classes). In our case we will have 2 classes: *pass* and *fail*.

For the following training and evaluation of the models we used roughly 250 labeled vectors. This should be enough to train different simple classifiers with sufficient accuracy on other unlabeled vectors.

```
 1 mins = []
 2 medians = []
 3 maxes = []
 4 q1s = []
 5 q3s = []
 6
 7 # Iterate over the thread result XML nodes of single benchmark operation
 8 for res in cpresult:
 9    for basetarget in res:
10       if basetarget.tag == "base_result":
11          b_attr = basetarget.attrib
12       if basetarget.tag == "target_result":
13          t_attr = basetarget.attrib
14    try:
15       # Compute relative differences between base and target results
16       medians.append((float(t_attr["median"]) - float(b_attr["median"]))
17                   / float(b_attr["median"]))
18       mins.append((float(t_attr["median"]) - float(t_attr["min"]))
19                   / float(t_attr["median"]))
20       maxes.append((float(t_attr["median"]) - float(t_attr["max"]))
21                   / float(t_attr["median"]))
22       q1s.append((float(t_attr["median"]) - float(t_attr["first_q"]))
23                   / float(t_attr["median"]))
24       q3s.append((float(t_attr["median"]) - float(t_attr["third_q"]))
25                   / float(t_attr["median"]))
26    except:
27       print("ERROR in values in " + path)
28
29 vector = []
30 # Add min, med and max value from each array of differences
31 for a~in [medians, mins, maxes, q1s, q3s]:
32    vector.append(min(a))
33    vector.append(median(a))
34    vector.append(max(a))
35
36 # Square root all element
37 vector = list(map(lambda x: math.sqrt(float(x))
38              if float(x) > 0
39              else -math.sqrt(abs(float(x))), vector)))
```

Figure 6.5: Python code showing reading and preprocessing of a vector from single comparison node parsed by ElementTree library for working with XML files. On the line 8 is iteration through comparisons of benchmark operations. On the line 9 the code looks for child nodes with base and target results. In the try block starting on the line 14 are computed relative differences of the statistical values. On the line 29 is are added minimum, median and maximum values from each array of relative differences of statistical values. The last operation on the line 34 applies square root on each element of the vector.

6.6.1 Validation and metrics

To compare success rate of different classification models we need a method to evaluate accuracy of predicting new data. Testing the model on data that we used for learning can lead to *overfitting*. Overfitting means that the model precisely predicts data it has already seen, but fails to predict any new data it has not seen yet. To avoid overfitting we use *cross-validation* method to test the model. It removes part of data from training set and uses them as testing set for evaluating the prediction.

For splitting the training data we use *k-fold* method. This method splits the original dataset to k groups. One of groups is used as testing set and the rest $k-1$ groups are left for training. This way we get k different dataset for evaluation of our classifiers.

As implementation of k-fold cross-validation we use `RepeatedStratifiedKFold` class from scikit-learn library with the parameter k set to 4. It will split the original dataset to 4 groups with same ratio of pass and fail labels repeating this process with differently shuffled data. Setting number of folds to 4 gives us enough data both for training and for the evaluation.

As metric for rating of predicted values we use *accuracy* scoring. Predicted results are evaluated as 1 or 0 if the predicted class is the same as the reference class from manual labeling or not. Averaging those scores from the whole testing set will give us ratio of correctly predicted vectors. In the following we will describe each tested classifier.

6.6.2 k-nearest neighbors classifier

This classifier does not construct any generalized model, but works with the whole training data. The decision is then made by voting of the k nearest neighbors.

The learning process consists only of building data structure for more efficient search through the dataset. More computations come with the prediction when the algorithm must go through the learning dataset and find k nearest vectors.

Accuracy of the model highly depends on the parameter k. Higher values suppress noise in dataset, but can smoothen the boundaries of classes too much. This behavior can be seen in Figure 6.6.

Scikit-learn implements k-NN classifier in `KNeighborsClassifier` class. Next to setting the k number it provides parameter for weighting votes. First option is uniform voting where each of the k neighbors have the same weight of the vote. The second option weights votes based on the distance from queried vector. The weighted option provides better results for our dataset as can be seen in Figure 6.6.

6.6.3 Linear logistic regression classifier

Linear model tries to separate vectors of two different classes to two spaces using a line, plane, hyperplane, etc. depending on the dimensionality of the vectors. It works by multiplying queried feature vector by learned set of weights. Thanks to this simplicity it scales very well for large amount of vectors and features.

The linear logistic regression classifier is implemented by scikit-learn in `LogisticRegression` class. We will use it with `class_weight` parameter set to *balanced*, because the linear classifier keeps the probabilities of classes favoring the by count of vectors it got for learning.

In Figure 6.4 we can see that our data are linearly separable and the linear logistic regression should be able to fit our dataset with good results.

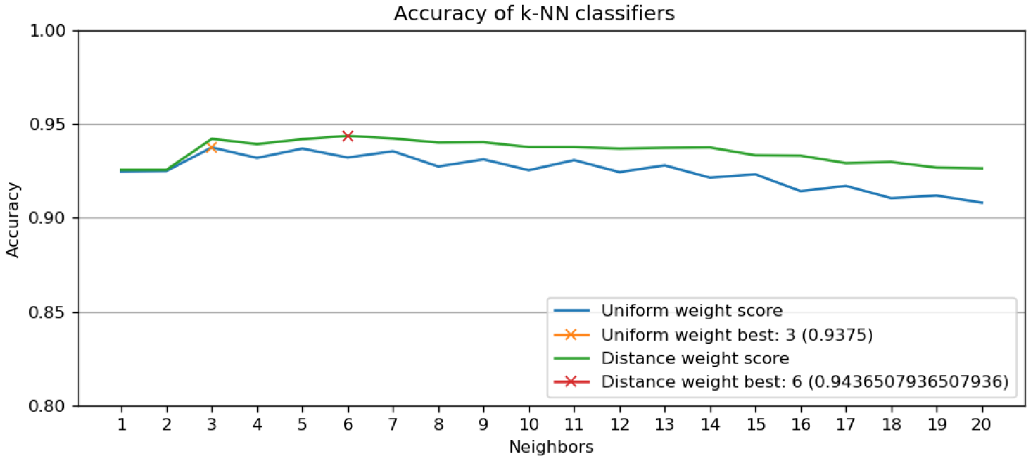


Figure 6.6: Accuracy of k-nearest neighbors classifier on available training data with different parameters. On x axis is number of k-nearest neighbors that vote for the class of queried vector. The two plotted lines differ in weighting of the votes. The first takes all votes with uniform weight and the second weights the votes by distance of the neighbors from the queried vector.

6.6.4 Decision tree classifier

Decision of this classifier is based on decision tree built on training data. In each node the tree looks on one component of the queried vector and by comparison with learned threshold the tree decides to which branch it will continue. Each leaf of the tree contains label that is returned for the queried vector which led to the leaf.

Learning of the model involves the construction of the decision tree. The construction looks for the best component of vector for splitting the dataset by classes. It creates a decision node and splits the training dataset based on the selected component to the decision branches. This process is recursively repeated with each part of the split dataset until the split contains vectors of single class (purity of the node is 1) or any additional restrictive condition is met.

Main advantage of the decision tree is its transparency and easy visualization of the model. In Figure 6.7 is an example visualization of the decision tree constrained by depth and samples count in leaf nodes built with our dataset of performance comparisons.

Disadvantage of the decision tree is sensitivity to noisy data and high chance of overfitting, because the tree tries to fit the whole dataset and has 100% prediction accuracy on already seen data. This issue can be suppressed by reducing maximum depth of the tree, setting minimal amount of samples required to split impure node into another decision.

6.6.5 Forest of randomized trees classifiers

The forest of randomized trees classifier extends the decision tree classification by creating a set of decision tree classifiers with small portion of randomness introduced in the creation of the decision trees. Output of the classifier is decided by voting of the decision tree classifiers. This modification compensates the overfitting of single decision tree classifiers. Although the amount of trees slightly increases the accuracy of the prediction, the duration of learning of the model and prediction rises linearly with the amount of trees.

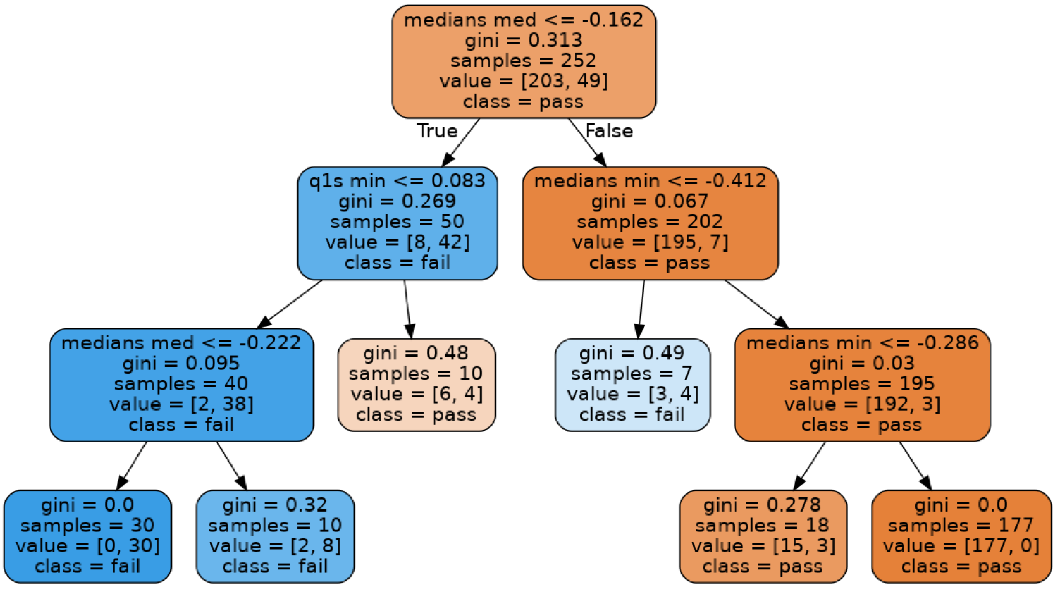


Figure 6.7: Decision tree created by Decision tree classifier and rendered by Graphviz library. It shows decision tree generated on our performance comparison data constrained by depth and samples count in leaf node. First line of non-leaf node shows the condition for choosing the next decision branch. *gini* property of node shows its impurity – proportion of training data not fitting the class of the node. Leaf nodes have 0 impurity because they contain vectors of single class. Parameters *samples* and *value* show the amount of training vectors for the subtree and their division to pass and fail classes.

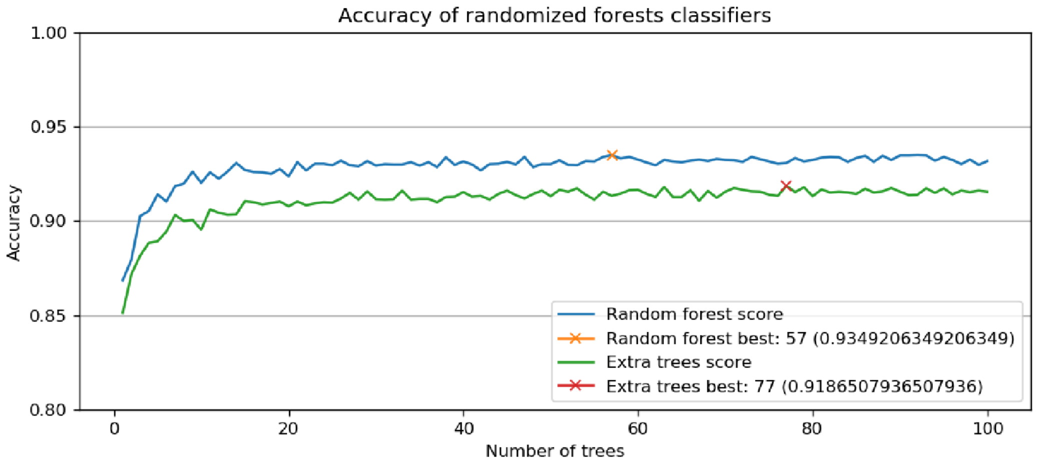


Figure 6.8: Accuracy of randomized forests classifiers on available training dataset with different parameters. On x axis is number of trees that vote for the class of queried vector. The first plotted line represents the Random forest method and the second the Extra trees method of building the trees.

The scikit-learn library provides two implementation of classifiers based on randomized trees. The first is `RandomForestClassifier` class. The dataset for each tree is resampled with replacement creating slightly reduced and different dataset. When building a decision tree, the component used for comparison in the node is not the best for splitting, but randomly chosen.

The second is extremely randomized trees method implemented in `ExtraTreesClassifier` class. This implementation extends the randomness by choosing more thresholds for decision randomly and selecting the best instead of computing the most discriminative one. This method helps to reduce variance of the model at the cost of higher bias.

Comparison of these two implementations is in Figure 6.8. The RandomForestClassifier has slightly better accuracy than ExtraTreesClassifier. The accuracy also rises much slower compared to number of the trees in the model.

6.7 Evaluation of classifiers

For the evaluation and comparison of proposed classifiers we will use these configurations:

- **k-nearest neighbors** with $k = 3$ and uniform vote weight.
- **k-nearest neighbors** with $k = 6$ and vote weight based on distance.
- **Linear regression** with balanced weight of the classes.
- **Decision tree** with minimum of 16 samples in each leaf node.
- **Random forest** with 100 trees.
- **Extra trees** with 100 trees.

Precision of the classifiers will be evaluated using k-fold cross-validation described in Subsection 6.6.1.

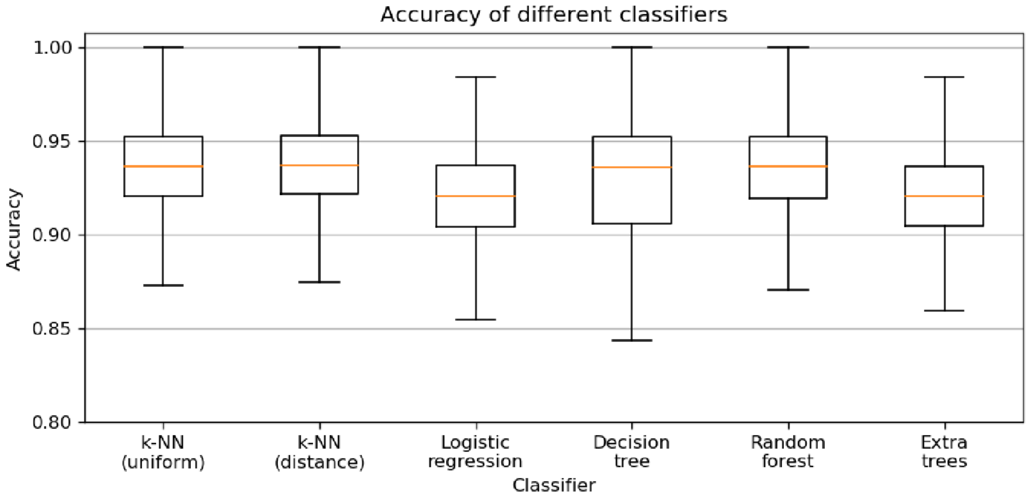


Figure 6.9: Comparison of accuracy of different implemented classifiers listed in Section 6.6. The results come from repeated k-fold cross-validation by splitting the dataset to train and test sets using ratio 3:1. The exact results are listed in Section 6.7

Results of the classifiers evaluation are visualized in Figure 6.9. Exact median values of classification precision on testing datasets with deviation of measurement are listed below:

```
k-NN (uniform):
  accuracy: 93.63% (+/- 4.96%)
k-NN (distance):
  accuracy: 94.29% (+/- 4.66%)
Logistic regression:
  accuracy: 92.05% (+/- 5.72%)
Decision tree:
  accuracy: 92.97% (+/- 5.82%)
Random forest:
  accuracy: 93.13% (+/- 5.66%)
Extra trees:
  accuracy: 91.85% (+/- 5.59%)
```

As the best classifiers seems to be *k-nearest neighbors* classifiers and *random forest* classifiers. Now any of these classifiers is fine to use for the automatic detection of performance regression, but with more manually labeled data will be the evaluation repeated and the selection of classifier reconsidered.

Chapter 7

Conclusion

In this work we described the performance testing of Linux kernel scheduler in Red Hat, Inc. company. The process of checking for performance regressions in new versions of kernels for Red Hat Enterprise Linux consist of measuring the performance of the kernel using benchmarks, storage of the results and visualization of performance comparisons between the targeted kernel version and reference the base version.

This work proposes a new way of visualization of performance results called *Timelines* focused on long-term performance development of multiple Linux kernels. The timelines were already used to compare the stability of operations from used benchmarks to create reduced test scenario with the more stable operations for less important tests with shorter run time. Compared to original scenario with estimated run time of 24 hours the shorter variant took only one third of the time. Besides the generated timelines, reports are continuously used to examine kernel performance through longer period of time.

Furthermore, this thesis proposed an utilization of machine learning for automatic classification of performance measurement comparisons to reduce time one has to spend to look for performance regressions. The trained model is going to be integrated in the generator of performance comparison reports to mark the comparison as passed or failed (i.e. containing performance regression). This will reduce time of examining dozens of reports from each new tested kernel and allow more time to work on new features of the generator. With more manually labeled data for learning the classifiers will be reevaluated and compared again. In case of better results the original model will be replaced with the better trained one.

Although the proposed timeline graphs reports and automatic detection of performance degradation push the effectivity of working with performance results forward, their potential is not depleted yet. Future addition of Jinja2[1] template system to timeline report generator will make the code much cleaner. The performance degradation classifiers will be reevaluated after obtaining larger manually labeled dataset and the automatic classification will be included to timeline reports to highlight performance regressions and watch precision of classification on new real data.

[1] http://jinja.pocoo.org

Bibliography

[1] Bailey, D. H.; Barszcz, E.; Barton, J. T.; et al.: The nas parallel benchmarks. Technical report. The International Journal of Supercomputer Applications. 1991.

[2] Bovet, D.; Cesati, M.: *Understanding The Linux Kernel*. Oreilly & Associates Inc. 2005. ISBN 0596005652.

[3] Corbet, J.: Coscheduling: simultaneous scheduling in control groups. *LWN.net*. Retrieved from: https://lwn.net/Articles/764482/

[4] Dongarra, J. J.; Luszczek, P.; Petitet, A.: The LINPACK benchmark: Past, present, and future. 2002.

[5] Gregg, B.: Active Benchmarking. http://www.brendangregg.com/activebenchmarking.html. accessed: 2019-01-23.

[6] Hewlett-Packard: Best Practices When DeployingLinux on the HP ProLiant DL980. https://lwn.net/Articles/764482/. accessed: 2019-04-24.

[7] Highsoft AS: Highcharts General Documentation. https://www.highcharts.com/docs/. accessed: 2019-05-08.

[8] Lozi, J.-P.; Lepers, B.; Funston, J.; et al.: The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. New York, NY, USA: ACM. 2016. ISBN 978-1-4503-4240-7. pp. 1:1–1:16. doi:10.1145/2901318.2901326.
Retrieved from: http://doi.acm.org/10.1145/2901318.2901326

[9] McCalpin, Ph.D., J. D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/. accessed: 2019-01-23.

[10] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; et al.: Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. vol. 12. 2011: pp. 2825–2830.

[11] Standard Performance Evaluation Corporation: SPECjbb2005 User's Guide. https://www.spec.org/jbb2005/docs/UserGuide.html. accessed: 2019-01-23.

[12] Standard Performance Evaluation Corporation: SPECjvm2008 User's Guide. https://www.spec.org/jvm2008/docs/UserGuide.html. accessed: 2019-01-23.

Appendix A

Content of attached CD

- **timelines/** directory contains Python source code of timeline reports generator with several examples of results.
- **classification/** directory contains Python scripts used for preprocessing of data for machine learning and for plotting of the graphs in Chapter 6.
- **text/** directory contains LaTeXsource code of this thesis.
- **README** file contains description of the content of this CD.
- **LICENSE** file contains license of the source code on this CD.