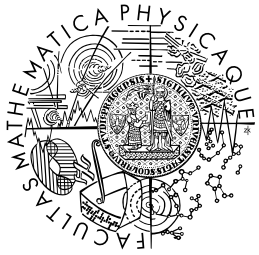

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Buchar

Load Balancing in Evaluation Systems for Programming Assignments

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Martin Kruliš, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to express my thanks to my supervisor, RNDr. Martin Kruliš, Ph.D., for his guidance in writing this thesis, and the patience and valuable knowledge he demonstrated on many occasions. I also highly appreciate that the Department of Software Engineering provided me with hardware for my experiments.

The ReCodEx team, namely Mgr. Martin Polanka, Bc. Šimon Rozsíval and Mgr. Petr Stefan, also deserves a honorable mention for designing and building the system that both inspired my research and served as a foundation for testing my ideas. I am grateful for the work we did together, the things we learned and the good times we had.

Finally, I wish to thank my friends and family, who stood by me and supported me during my studies and helped me stay on track.

Title: Load Balancing in Evaluation Systems for Programming Assignments

Author: Jan Buchar

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D., Department of Software Engineering

Abstract: Systems for automated evaluation of assignments are a valuable aid for both teachers of programming courses and their students. The objective of this thesis is to examine the possibilities of deploying such systems in a large-scale distributed environment and the challenges of such endeavors. A sizable part of the requirements comes from experience with ReCodEx – an assignment evaluation system developed at the department of the supervisor.

Modern server multi-core processors provide considerable computing power that can be used for assignment evaluation. However, parallel measurements can interfere with each other. This causes unstable results, which detriments the fairness of grading. Isolation (sandboxing) technologies can cause similar effects. We measure both of these influences and use the results to determine to what degree can multi-core processors be exploited.

The problem of efficient distribution of work between multiple evaluation workers is complementary to that of utilizing multi-core machines. We survey scheduling algorithms and design an experiment to compare their performance. Additionally, we examine the possibility of leveraging container technologies to simplify the deployment of software required for evaluation. This leads to both a smaller administration overhead and a less complex structure of the pool of evaluation workers, which simplifies the task of the scheduler.

Keywords: Load balancing Scheduling Sandboxing Virtualization Cloud computing

Contents

Introduction

Automated evaluation of programming assignments is a valuable aid in the teaching of programming. It reduces the time students need to wait for feedback after a submission. In addition, it is possible to grant students multiple attempts to solve an assignment with negligible added costs. That, in turn, allows the teachers to assign more demanding tasks that help developing programming skills more efficiently.

The automation also facilitates filtering out submissions that do not meet objective criteria prescribed by the teacher, such as being syntactically correct, yielding correct outputs for example inputs or finishing in a specified amount of time. This gives the teachers an opportunity to focus on qualities that are difficult to assess automatically, such as good object-oriented design or code readability, without having to bother with repetitive tasks like compiling submitted source code and checking the basic functionality on example inputs and outputs.

This thesis aims to design a system for programming assignment evaluation that is flexible enough to work efficiently both on physical, multiprocessor servers and on virtual machines provided by a cloud platform. Such system could be used to create a community-driven programmer training platform, thus making education in programming available to the whole world. Additionally, universities could deploy instances of the system using their own hardware and possibly customizing it.

We assess the viability of such deployments in the context of ReCodEx – a system for evaluation of programming assignments developed at the department of the supervisor. There are multiple properties inherent to the problem of automated assignment evaluation that complicate efficiently scaling the system. For example, many assignments rely on time measurements being stable to a reasonable degree – without that, it is impossible to reliably test if an algorithm is implemented efficiently. Furthermore, it is necessary to isolate the submitted programs to prevent malicious or extremely inefficient code from bringing the system down. Various ways of doing this could also impact the results of our measurements.

We examine the possibilities of exploiting dedicated (private) multiprocessor server machines for evaluation of student submissions. Stressing a dedicated server too much with parallel measurements might lead to unpredictable results. Despite that, not using multiprocessing at all would waste the potential of modern server computers.

The influence of multiple technologies for isolation and secure execution of programs submitted by students on the stability of measurement results is also examined. Among those, both containers and virtual machines are represented. These technologies are necessary for the robustness of the system, but some of them might also help stabilize time measurements as a side effect and thus allow for more efficient usage of multiprocessor hardware.

Next, we include a comparison of load balancing strategies in the context of programming assignment solution evaluation, as the choice of a load balancing algorithm greatly affects the overall throughput of the whole system. The comparison also takes into account a possibly heterogeneous computing environ-

ment (e.g., both physical and virtual machines with different capabilities) and the on-demand scaling features of current virtual infrastructure providers.

The last part of the thesis deals with the possibility of leveraging container technologies in other ways than to run submissions in isolation. They could also be used to simplify the deployment and maintenance of runtime environments for various programming languages. Another possible use case is supporting user-defined software stacks by allowing exercise authors to modify the runtime environments, for example by installing additional libraries that will not be available for other exercises, without introducing overhead for the system administrators.

The thesis is structured in the following way. In Chapter 1, we introduce the problem of automated programming assignment evaluation in detail. In Chapter 2, the stability of time measurements is evaluated with various isolation technologies and degrees of parallelism. We use the results to determine how many parallel measurements can be used on a single machine with multiple CPUs, and whether using isolation techniques is beneficial or detrimental to the stability. Chapter 3 categorizes our specific variant of the scheduling problem in the context of prior research in this area, and then presents an experiment that compares the performance of existing algorithms. Chapter 4 examines the possibilities of using containers to simplify the deployment of software environments required for evaluation. Key parts of the functionality is implemented and a short benchmark is presented. Finally, Chapter 5 analyzes the integration of on-demand scaling into an assignment evaluation system.

1. Assignment Evaluation

In this chapter, we outline the specifics of automated evaluation of programming assignments from the perspective of load balancing and large-scale deployments. We also introduce ReCodEx – an evaluation system that motivated this thesis. The chapter also aims to explain the most important concepts of the problematics for future reference.

1.1 Analysis of Requirements

This section aims to list the features that are required in a programming assignment evaluation system so that we can contrast these to the features implemented by ReCodEx. The requirements were gathered by surveying various assignment evaluation systems and environments for the management of programming contests (a closely related topic), and also during the operation of ReCodEx.

1.1.1 Correctness

There are many objective qualities of a computer program that can be assessed automatically. The most important and obvious one is whether the program responds correctly to a set of test inputs. In the simplest case, we only need a set of input files and a set of corresponding output files that can be compared with the actual output of the program.

The more complicated cases involve situations with multiple possible solutions (such as finding the shortest path in a graph) or with a non-binary way of determining correctness, where the program output is assigned a decimal number between 0 (absolutely incorrect) and 1 (absolutely correct). An example of such correctness measures could be the accuracy of predictions made by a neural network or the ratio of a compression algorithm.

1.1.2 Execution Time

Execution time is another important evaluation criterion. There can be solutions that are logically correct, but require too much time to yield the results. Also, some solutions might get stuck in an infinite loop or a deadlock for some inputs. The second case is especially problematic as it could render the service unavailable until an administrator terminates the evaluation. For this reason, it is necessary to both measure and limit the execution times.

There are two principal ways of measuring execution time measurement. The first one is to measure CPU time – the amount of time used for the actual execution of the program, calculated from the elapsed number of CPU cycles. The other one is measuring wall clock time, which is the length of the interval between the start and the end of the program execution. The main difference from CPU time is that it also includes time used by system calls, waiting for I/O operations, and sometimes (depending on implementation details) also the time when the program was not running at all due to context switching.

It is critical to limit the wall clock time in every evaluation. Otherwise, a program that sleeps indefinitely using a system call would never be terminated. A limit on CPU time can also prove useful (even though the CPU time is also affected by a wall clock time limit). It can be used to limit the execution time more precisely for CPU-intensive exercises, or to provide the same amount of computation time to all solutions in an exercise that allows using parallel computing.

Ideally, all measurements of execution time should be fair and reproducible (this applies to other performance metrics as well, but execution time is notably unstable). Without this, teachers could not depend on the measurement results for grading, because the resulting grade could be different each time.

1.1.3 Resource Utilization

Memory usage is another important measure of the efficiency of a program. While it sometimes cannot be directly controlled by the programmer (due to garbage collection and implementation specifics of memory allocators), a high memory usage can be an indicator of an inefficient algorithm. Moreover, a malfunctioning program could bring down the evaluation computer by allocating too much memory for other programs to function correctly.

Apart from time and memory, which are essential performance metrics, there are other system resources whose usage shall be limited (or disabled in some cases) – for example disk space and network bandwidth.

1.1.4 Additional Metrics of Code Quality

Static code analysis can also provide valuable insights about code quality. For example, it might be useful to filter out solutions that do not handle all exceptions that can be thrown in the program for some languages and exercise types. Another example is ensuring that the source code adheres to a specified coding style guideline. This kind of functionality is sometimes provided by compilers, and also by specialized utilities called linters.

Other tools can be used during the runtime of the program for various reasons. We could for example check for memory leaks with Valgrind (minor ones might evade memory limits), detect invalid usage of pointers and arrays with mudflap or report performance metrics in environments that use a virtual machine, such as Java (where JVM instrumentation could possibly be used).

1.1.5 Isolation of Executed Code

It is critical to guarantee that submitted code is run in an isolated environment. An untrusted program should not be able to exploit the host system (the system performing the evaluation), for example by accessing its files or by using interprocess communication (such as shared memory or UNIX signals) to communicate with system daemons. Such activities might even lead to a takeover of the host system.

Connecting to other evaluated programs must also be prohibited. Otherwise, student submissions could for example read output files of other programs to bypass limits on processing time and memory.

1.1.6 Resilience

Despite isolation technologies being used, it is still possible for parts of the system to malfunction. This is even more of a problem in distributed systems, where network errors can cause problems. The system must be designed to recover gracefully in these cases. In particular, evaluations that failed due to an external error (i.e., not because the solution is incorrect) should be retried without the need for an intervention from an administrator. Of course, if a submission is in fact impossible to evaluate, we must limit these retries so that the evaluation does not proceed infinitely.

1.1.7 Scalability

We understand scalability as the ability of the system to adapt to a growing or declining number of clients by adding or removing resources. In a programming assignment evaluation system, the driving factor for scaling is almost exclusively the number of submissions.

While automatic on-demand scaling is the ultimate goal, supporting manual scaling (aided by an administrator) could also be sufficient. In fact, the main benefit of automatic scaling is being able to react to sudden peaks in the number of submissions.

1.1.8 Latency and Throughput

Fast feedback is one of the critical selling points for automating the process of evaluation of programming assignments. Therefore, it is necessary that the system schedules the received submissions in a way that keeps the latency within reasonable bounds. The system should maintain an overall low latency even with an increasing number of evaluations performed at the same time.

The scheduling and on-demand scaling (or more broadly, resource management) policies should also take throughput in account. The system must be able to serve hundreds of clients at the same moment.

1.1.9 Extensibility

The system should allow adding support for new languages without significant changes to the code base and without overhead for the administrators. Also, extending the runtime environments with libraries should be possible on a per-exercise basis, as installing a library system-wide would make it available even for exercises where using it is not desirable.

1.2 ReCodEx

ReCodEx is a system for evaluation of programming assignments developed at the Faculty of Mathematics and Physics of the Charles University. It was first deployed in 2016, but it still undergoes active development. Since it is relatively modern and also well known to the author of the thesis, it will be used as a reference for reasoning about the implementation of features discussed in this text.

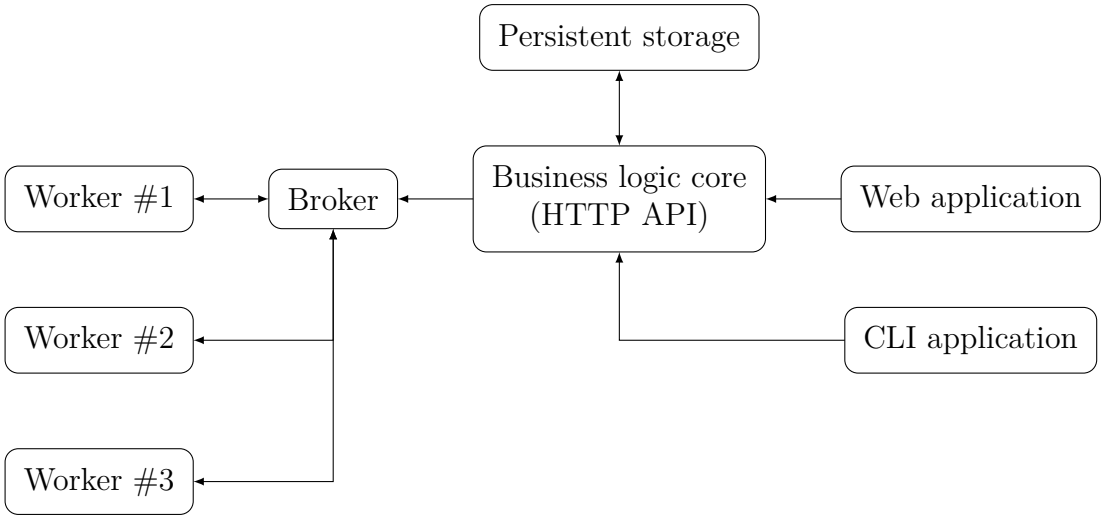

Figure 1.1: A simplified diagram of the components of the ReCodEx code examiner

We can also expect that the results of this thesis will influence the development of ReCodEx and that some parts will become direct contributions to the project.

1.2.1 System Components

The system is divided into multiple independent components depicted in Figure 1.1. The functionality of the system is exposed to users by an HTTP API provided by a business logic core, which is run as a CGI-like application in a web server. The business logic requires persistent storage of many kinds of data ranging from user accounts to test inputs for exercises and evaluation results. A combination of a relational database and plain file system storage is used. Thanks to the wide adoption of the HTTP protocol, the API can be used by a variety of client applications. Currently, ReCodEx has a web application and a low-level command line frontend.

The broker is a component that is critical for the evaluation of assignment solutions. It receives evaluation jobs from the core and forwards them to workers. It is also responsible for monitoring the status of the workers, balancing their loads efficiently and handling outages and evaluation failures.

Each worker evaluates one job at a time in a secure isolated environment. However, multiple workers can be deployed on a single physical machine for a more efficient usage of the hardware (typically multiple CPUs).

1.2.2 Worker Selection

The pool of worker machines in ReCodEx can be fairly diverse. There can be machines that differ in hardware (newer and faster machines can be added while the old ones remain), operating systems (most workers use GNU/Linux, but future assignments might require Microsoft Windows, for example), available software (support for some languages does not have to be installed on every worker machine), and possibly in other characteristics.

Each worker machine advertises its properties when it is registered with the broker. Evaluation jobs are also assigned a set of headers – a structured specification of the requirements on the worker that is going to process the job – by the business logic core when a solution to an assignment is received. It is a responsibility of the broker to select an appropriate worker.

While this provides a great deal of flexibility, it also presents a challenge in efficient scheduling of evaluation jobs. Currently, the broker maintains a separate queue of pending jobs for each worker and incoming jobs are assigned to these queues in a simple round-robin fashion.

1.2.3 Secure Execution of Submissions

ReCodEx uses the `isolate`[1] sandbox to ensure that code submitted by students is executed in a secure and isolated environment and that their usage of resources is limited. The sandbox is controlled by the `isolate` command line utility, which is executed as a subprocess by the worker daemon.

A separate instance of the sandbox is used for each stage of the evaluation process where untrusted code is involved. In particular, this means compilation of source codes and running the resulting program with test inputs. Files that should be kept between the stages (typically compiled binaries) must be copied by the worker (the files that are supposed to be copied are specified by the configuration of the exercise).

1.2.4 Judging Correctness of Output

ReCodEx supports various ways of judging the correctness of the output of a submission using judges – programs that read the output file and assign it a rating between 0 and 1. There are multiple kinds of correctness rating implemented by built-in judges and it is also possible to upload a custom judging program for specialized exercises.

1.2.5 Description of Evaluation Jobs

When the core receives a submission for evaluation, it takes the exercise configuration prepared by the author and the submitted files and transforms them into a structured file called the job configuration (which is then used by the worker).

The job configuration is a YAML-encoded object that contains job metadata and, more importantly, the instructions on evaluating the submission. The instructions are a set of atomic tasks of various types. One group of tasks are internal tasks, which are implemented by the worker itself. Examples of this group are downloading a file from the persistent storage, extracting an archive or copying a file from one place to another. The other group are external tasks. These involve running a program in a sandbox (currently, only `isolate` is supported) with a set of limits and with measurements taking place. This is mainly used for compilation and execution of submitted code.

The set of tasks is not in a fixed order. Instead, dependencies are specified explicitly so that that the tasks form a directed acyclic graph. If a task fails, the worker cancels only its dependees and not the whole evaluation. This allows

for a great deal of flexibility in use cases such as conditional evaluation (e.g., a submission must pass at least one of two tests).

1.2.6 Language Support in Workers

In order to support a programming language, the machine that runs the worker must provide some utilities – typically a compiler or an interpreter. Currently, the core relies on the diligence of the administrators that maintain the worker machine – each worker must advertise the correct runtime environment headers (configured manually) and have the utilities required by their environments installed at the exact locations expected by the core. Also, adding a new language requires a (rather minor) change in the business logic code responsible for the generation of job configuration.

1.2.7 Result Consistency on Heterogeneous Hardware

The workers can run on completely unrelated machines with different hardware specifications. This has numerous advantages, most importantly that we can gradually replace obsolete machines with new ones and we can add specialized machines for some exercises (e.g., multiprocessor computers for parallel programming).

A drawback of this fact is that measurements on different hardware will likely have different results – a faster CPU will execute a test faster than a slower one. If not addressed, this would greatly impact the fairness of grading.

The solution chosen by ReCodEx is defining hardware groups – manually configured string identifiers shared by machines with similar hardware specifications, and having a separate set of limits for each hardware group allowed by an exercise.

1.3 Requirements Fulfilled by ReCodEx

In this section, we examine which requirements from Section 1.1 are satisfied by ReCodEx and which are not. We shall then evaluate the latter group and clarify which of those will be addressed by this thesis.

The requirement on detecting incorrect solutions (Section 1.1.1) can be considered satisfied. ReCodEx supports a wide range of judges that allow testing the correctness in many different ways. One thing that is missing is the support for interactive tasks where the solution communicates with another program.

Thanks to `isolate`, ReCodEx can measure and limit the usage of a plethora of resources, including CPU time, wall clock time, memory and disk usage (Sections 1.1.2 and 1.1.3). However, we do not know if the stability of measurements cannot be influenced by the isolation (or by multiple parallel measurements sharing the hardware).

The submitted code is also run in isolation from the rest of the host system and from other solutions (Section 1.1.5). The implementation makes it appear to the submission that it runs as the only process in its own operating system that includes a file system, inter-process communication and network communication. Of course, unless explicitly allowed, the solution cannot reach any other programs using these facilities.

ReCodEx is tolerant to failures in evaluation, even when they result in a crash of the worker machine (Section 1.1.6). In such cases, the broker reassigns the evaluation to another worker (with a limit on the number of reassignments). Evaluation jobs are not lost even in the case of a broker breakdown. They are stored persistently by the system core and once the broker becomes available again, they can be resubmitted.

ReCodEx can be easily scaled manually by adding more worker machines (Section 1.1.7). This also includes adding more powerful hardware over time. However, there are two problems that should be addressed. First, manual scaling cannot deal with sudden peaks in usage efficiently. Second, we do not know if the load balancing algorithm implemented by the broker is good enough to use the additional worker machines efficiently. As of now, ReCodEx uses 17 workers, 8 of them that are general purpose and 9 that are reserved for specific subjects.

The load balancing algorithm also has a notable influence on the latency and throughput of the system (Section 1.1.8). Therefore, its efficiency must be measured and compared to alternatives.

The last two remaining requirements to examine are extensibility (Section 1.1.9) and support for additional code quality metrics (Section 1.1.4). Thanks to the general job configuration format used by the worker, the only thing left to satisfy the second requirement is being able to provide the quality checking software. The ReCodEx workers can use any software installed on the host machine, which partially satisfies both of the requirements. However, the software has to be installed first and the system core has to be modified to emit job configurations that use it. This would introduce a notable administration overhead if we needed to support per-exercise runtime environments.

1.3.1 Summary

From our examination of the requirements and the reality of ReCodEx, we conclude that a number of topics has to be researched. The stability of measurements has to be measured when `isolate` is used and when multiple measurements are performed on the same hardware.

Furthermore, the efficiency of the load balancing algorithm must be examined, because it affects the latency, throughput, and scalability of the system. Also the viability of on-demand scaling should be assessed.

The last obstacle on the path to a large scale deployment of ReCodEx is the overhead of adding and extending runtime environments, and it could be removed by using container technologies.

2. Stability of Measurements

In this chapter, we examine the influence of various factors on the stability of time measurements in a system that evaluates programming assignments. Informally, we define a stable measurement as one that yields the same or very similar value each time it is repeated (provided that the input data is the same and that the measured program is deterministic).

This property is crucial in assignments which require students to submit programs that are not only correct, but also efficient in terms of execution time. For many problems, the benefit of an efficient algorithm manifests only on large inputs. On the other hand, it is important that the evaluation takes as little time as possible so the system can provide feedback quickly. If the measurements are not stable, the difference between efficient and inefficient solutions might be visible only after we test them with larger inputs, thus counteracting the instability. Stable measurements allow us to keep the input data small, which in turn makes the response period of the system shorter.

This idea is illustrated in Figure 2.1. For `n=25`, the 5% relative error margins do not overlap, while the 20% ones do. The value of `n` has to be increased over 50 before a clear distinction can be made with 20% relative error margins.

We examine two groups of factors that influence measurement stability. In the first one, there are various kinds of system load on the hardware performing the measurements. In the second one, there are technologies that allow us to run untrusted code in a controlled environment (e.g., process isolation or virtualization). Even though these technologies are required for the system to function securely, some of them might also help with mitigating the influence of the system load.

2.1 Analysis

To explore the influence of aforementioned factors on measurement stability, we shall measure a reasonable set of workloads under different execution setups with varying levels of system load and different isolation technologies.

2.1.1 Inherent Stability-influencing Factors

The design of modern computers introduced numerous optimizations to increase the overall throughput of the system. Conversely, reproducible measurement of execution time is not a primary objective in the design of a contemporary processor. Therefore, there are many factors that increase the performance of a computer at the cost of introducing a certain level of non-determinism to time measurements.

Caching has a large influence on the performance of some operations and it happens on many levels during the execution of a program. In the case of evaluation of programming assignments, CPU cache and disk cache are the most important.

CPU cache exists to speed up accesses to frequently used areas of memory and it also helps during sequential reads of data (although most modern processors

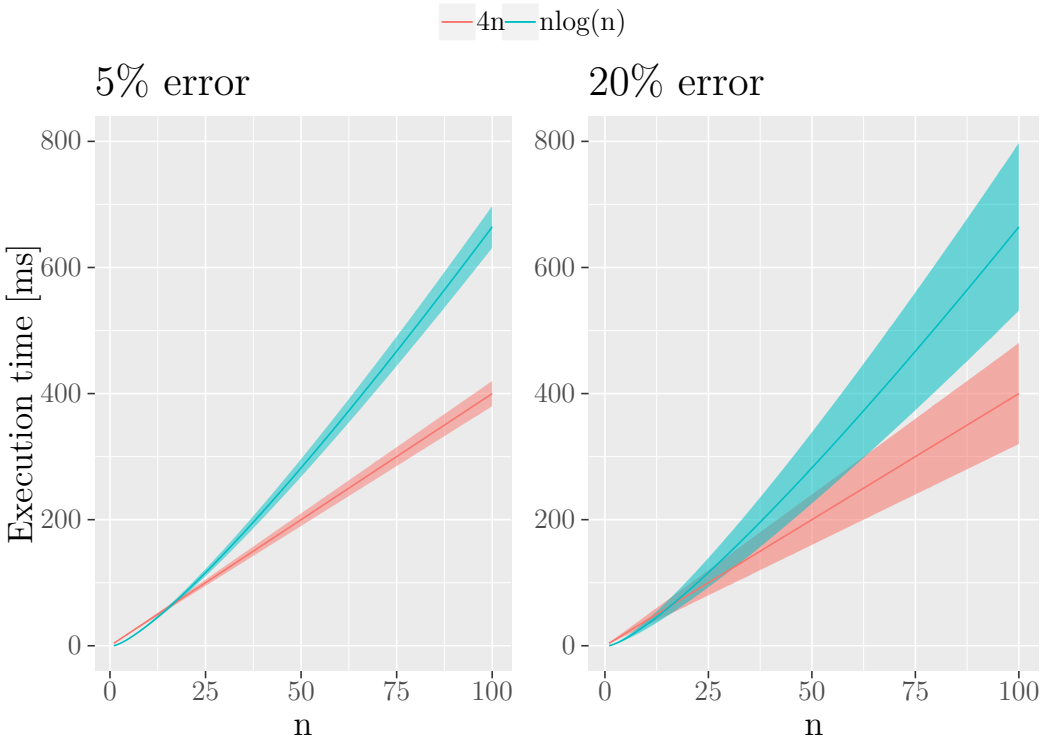

Figure 2.1: A comparison of the plots of two asymptotically distinct time complexity functions with 5% and 20% relative error margins outlined, where n is the input size

feature memory pre-fetchers whose influence is much larger in this case). Unfortunately, we have no control over the content of the cache when the program is launching. In addition, the cache is shared with other programs running on the machine, whose memory accesses cannot be controlled. Both of these facts make the time required for memory accesses less predictable, making the CPU cache a source of time measurement instability.

Disk cache (or page cache) is a mechanism for improving the speed of accesses to external memory (e.g., a HDD or SSD). Especially for HDDs, random accesses can be costly, which again introduces instability into time measurements. Although exercises that require a substantial usage of external storage (such as sorting in external memory) should not base the grading on precise time measurements, the effects of page cache might also manifest in reading an input file or when the measured binary itself is being loaded.

In a modern CPU, frequency scaling and power management (disabling inactive parts of the processor) take place to save energy when the system is idle. This becomes a problem when the system suddenly needs to start working after a period of inactivity. The frequency is typically not increased instantly, which can offset time measurements. Some of these features can be disabled to alleviate these effects.

Multiprocessing introduces another set of problems. Some parts of the CPU can become a bottleneck by not allowing simultaneous access for all cores that can be running in parallel – the memory controller is an important example of this.

Also, the processes can contend for the CPU cache in multiple ways. For example, when two processes run on the same core, they can access the L3 cache in a pattern that makes each of them force the cached addresses of the other process out. This can also happen when one process is suspended by the scheduler to allow another process to run or when it is migrated to another core.

Languages that feature JIT (just in time compilation), such as C# or Java, can also suffer from measurement instability because of this. It is common practice when benchmarking these languages to let the benchmark warm up before starting the actual measurements. This way, the JIT compiler has a chance to optimize the critical parts and we do not have to account for the overhead of both running unoptimized code and performing the optimizations when we evaluate the results. This practice can also help in environments without JIT by populating the CPU caches and pre-loading the binary into the page cache.

2.1.2 Isolation Technologies

The primary purpose of running submitted code in an isolated environment is to ensure that it will not damage the host system by excessively using its resources (e.g., memory or disk space), and that it will not bypass the restrictions on resource usage by communicating with the outside world (e.g., by reading results from other submissions or delegating work to network services). Additionally, some isolation technologies also provide accounting of resource usage, which is necessary for grading of submissions.

In this section, we survey existing isolation technologies and select a handful whose effects on the stability of measurements will be examined. By intuition, any additional isolation layer adds overhead and therefore might make the time measurements less predictable. On the other hand, the opposite might also be true – especially when there are multiple measurements running in parallel, process isolation could help stabilize the results.

UNIX chroot

The `chroot` system call (present in the UNIX specification since version 7, released in 1979) changes the root directory of the calling process[2], thus isolating it from the rest of the system and preventing it from accessing files not needed for assignment evaluation. Historically, this has been used as an additional layer of security for services that handle potentially dangerous input such as web servers.

Chroot itself however does neither limit resource usage nor provide accounting. Inter-process communication and network access are also not limited.

Debugger-based Isolation

`ptrace` is the UNIX interface primarily used by userspace debuggers. It can also be used to intercept system calls to achieve process isolation. By combining this with `chroot` and a resource limiting call such as `ulimit`, a full-fledged sandbox can be created.

The MO system for evaluation of submissions in programming contests features a sandbox like this[3]. The same sandbox was also used in CodEx (a pro-

gramming assignment evaluator released in 2006) and the CMS contest management system [4].

A notable problem of this approach is that it does not work well with multi-threaded programs. The `ptrace` interface only suspends the main thread on system calls. Therefore, a sandbox with multi-threading support would have to intercept calls that spawn new threads and start monitoring them too. Such approach would however introduce a host of new problems and attack surfaces. For example, it would be possible to spawn two threads, A and B, where A performs a permissible system call and B changes the context of thread A before the system call is actually executed, but after the sandbox is done inspecting it.

FreeBSD Jails

Jails[5] (featured since 2000) expand on the concept of chroots. In addition to confining a process to a part of the file system, they also provide network and process isolation and time, memory, and disk usage limits[6].

Many other UNIX systems also have their own implementations of jails, e.g., zones in Solaris or sysjail in OpenBSD and NetBSD.

Linux Containers

The Linux kernel supports creating containers – lightweight execution units that can be used for isolation and resource limiting.

Process isolation can be achieved using namespaces[7], a feature present in the kernel since 2006. These allow locking the process in an environment where communication methods such as networking or reading files seemingly works without restrictions, but the sandboxed process can only communicate with processes in the same namespaces (granular sharing is also possible to allow e.g., connecting to services over the Internet).

Resource limiting and usage accounting is implemented using control groups (cgroups)[8], merged into the kernel in 2007.

It can be reasoned that these measures should not have any noticeable overhead, at least compared to processes running in the global process namespace and cgroup – in modern versions of Linux, the same restriction mechanisms are used in any case, even when no isolation is desired and no additional namespaces and cgroups are created by the user.

Linux containers have been adopted by many projects, some of which we list here:

- **Docker**[9], which is the most prominent container technology as of now. It provides means for building images (templates for creating containers), transferring them between hosts via Docker Registry, creating containers based on the images and running programs in them. Docker is based on an open source project called Moby, which implements a number of specifications authored by the OCI (Open Containers Initiative)[10], mainly the Runtime Specification (which describes how to run a container from a local file system) and the Image Specification (which describes the format of image data and metadata). Over time, projects that provide alternative implementations for parts of the OCI-specified functionality have emerged, such

as Podman[11] or Buildah[12]. The main use case supported by Docker is deployment of applications or services as containers, as opposed to projects that use containers as lightweight virtual machines. This choice has many consequences in the way Docker is used – for example, data persistence must be set up explicitly by binding paths in the host file system into the container.

- **LXC**[13] (an abbreviation of "Linux Containers") provides a usage flow that is more similar to the traditional virtual machine computation model – a template is downloaded and used to create a container, which is a fully functional operating system that uses the kernel of the host. Users can then attach to this container and run commands inside it like they would in a virtual machine or a remote server. At the inception of the project, Docker used LXC as its backend for running containers.
- **Isolate**[1] is a minimalistic wrapper around cgroups, namespaces and other resource limiting facilities provided by Linux. It was designed for running and measuring resource usage of untrusted code in programming contests and homework assignment evaluation.
- **Singularity**[14] is an effort to bring user-defined software stacks to systems for high-performance computing. It uses Linux namespaces to isolate executed code. It can also integrate with resource managers such as Slurm, which is not a typical use case for Docker, for example. However, OCI images can be used as a base for Singularity images.
- **Charliecloud**[15] is a set of scripts for running Docker images on existing infrastructure with minimal alterations, without the Docker daemon itself. It does however use Linux namespaces for isolation.

As a side note, there were efforts to implement container support in Linux even before the inception of namespaces and cgroups. Possibly the most widely adopted one was OpenVZ[16] (released in 2005, based on commercial Virtuozzo from 2000). It shipped a modified Linux kernel that enabled container isolation and also provided hardware virtualization support in later versions.

Virtualization and Paravirtualization

Virtualization allows to run multiple guest operating systems on a single physical host without modifying them. The guests then operate under an illusion that they are running alone on a physical machine[17].

The virtualization is enabled by having a virtual machine monitor (commonly called the hypervisor) installed in the host system. There are multiple ways of running the code of the virtual machine, but in all of them, it is desirable to run as many instructions directly, without any intervention from the hypervisor. However, for some instructions, this is not possible – for example, memory access instructions can have a multitude of possible side effects, such as triggering memory-mapped IO or page table modifications. This can be handled in a variety of ways, like trapping these instructions and emulating them, dynamically translating them to different instructions or exploiting infrastructure for virtualization provided by CPU manufacturers.

Paravirtualization requires the guest operating system to be modified to avoid emulation of some instructions. For example, a block device driver can be im-

plemented by directly calling the virtual machine monitor on the host which can keep the data from an emulated drive in a file or in memory. Xen[18] is one of the most prominent paravirtualization technologies.

It is important to note that in some virtualization platforms (such as VirtualBox), the virtual machine monitor runs in userspace, which means it is also subject to resource management mechanisms such as process scheduling.

It is evident that these mechanisms can affect measurement stability. The effect can be either negative, because the virtualization could introduce additional non-deterministic factors into the measurements, or positive – the virtualized equivalents of IO operations, for example, might prove to be more stable than the actual operations.

The Selection for our Measurements

From the survey of possible approaches to process isolation, we have selected the following technologies for our measurements:

- Bare metal (**B**) – no isolation at all (used as a baseline value).
- Isolate (**I**) – a sandbox solution used in ReCodEx and other systems (such as CMS or Kattis[19])
- Docker (**D**) – the most popular container platform as of today.
- Isolate in Docker (**D+I**) – a combination that might be used to support user-supplied runtime environments in ReCodEx. Isolate might still be necessary to protect the insides of the container from the code supplied by students (an attacker that gains control of the container could e.g., report any grades they like to the rest of the system) and to measure resource usage.
- VirtualBox (**V**) – a readily available virtualization solution that does not need extensive setup. We will manage our VMs using Vagrant so that we can easily take measurements of other virtualization platforms in the future.
- Isolate in VirtualBox (**V+I**) – the reasoning for adding isolate is the same as with Docker.

An important deciding factor in the selection of isolation technologies was the adoption of GNU/Linux, both in the field of programming contests and internet servers in general. In addition, ReCodEx, while being built to also support measurements on Windows, primarily uses GNU/Linux. Unfortunately, this choice disqualifies technologies like FreeBSD jails. On the other hand, these are conceptually very similar to Linux containers.

The measured data will be compared to values measured on the bare metal. Measurements will also be performed with manually configured CPU affinities to see if such configuration has any effect on the stability of time measurements. Setting the affinity for VirtualBox VMs is very difficult when parallel processes are involved, so this setup will not be included in the experiment.

The Linux kernel also allows setting per-process NUMA affinity, which determines which memory nodes should be used by the process. Restricting a process to the memory node that belongs to the CPU where it is running is certainly reasonable. Since this restriction is the default policy in Linux[20], we will not measure the setup where the CPU affinity is already set explicitly. However, we

will experiment with setting the NUMA affinity without an explicit CPU affinity (i.e., restricting a process to a memory node and not to a CPU).

2.1.3 Execution Setups

There are multiple ways of simulating measurements on a machine where other processes are running. First, we can run multiple instances of the measurements of the same exercise type in parallel. While it might seem like an artificial situation, it is actually a likely scenario – it often happens that multiple students start submitting solutions to the same assignment at the same time (for example when the deadline is close). This execution setup type is called `parallel-homogeneous` in plots and measurement scripts.

Second, we can use a tool that generates system load with configurable characteristics. Such experiment does not imitate real traffic as well as the `parallel-homogeneous` variant, but the results might prove easier to interpret and reproduce. Moreover, the ability to configure the characteristics of the system load could help identify which kind of system load influences the measurement stability the most.

To implement this type of execution setups, we will use the `stress-ng`[21] utility, and along with that, we will run measurements of a single exercise type. In plots and measurement scripts, the names of these C setups start with `parallel-synth`.

In order to examine the behavior of the system under varying levels of system load, we will repeat the measurements with different amounts of workers running in parallel. The amounts of workers shall be chosen with regard to the topology of CPU cores so that they exercise all variants of cache utilization. For example, on a system with two dual-core CPUs where each physical core has two logical cores, we will want to run:

1) a single process,
2) two processes (each uses one CPU cache),
3) four processes (one per physical core, two pairs of processes will share the last level cache) and
4) eight processes (one process per logical core, i.e., exploiting the logical cores).

Launching more processes than there are logical cores might be an interesting experiment. Sadly, there is little value in it in for our research, because all these processes could not run in parallel at the same time and therefore, the total throughput would not increase. Such configurations would be viable if we included IO-bound workloads in our measurements – we could have more parallel measurements than there are CPU threads, some of which could run while other threads wait for IO.

The parallel workers will be launched using GNU parallel[22], a relatively lightweight utility that simplifies the task of launching the same process N times in parallel with a variable parameter. There are numerous alternatives to parallel with negligible differences, at least considering our use-case where we simply need to launch a fixed number of commands simultaneously on the same machine.

Nonetheless, we shall make sure that the measurements did in fact run in parallel in the evaluation of results.

Throughout the text, we understand "execution setup" as a union of execution setup type (e.g., `parallel-homogeneous`) and system load level (the number of parallel workers).

2.1.4 Choice of Measured Assignment Types

There are numerous types of programming assignments suitable for automated evaluation that differ in their characteristics and requirements. We mostly differentiate them by the bounding factor in their performance – the speed of the CPU, memory accesses or IO operations.

The performance of CPU-bound programs is primarily limited by the speed at which the processor can execute instructions. An example of this class are exercises that require students to perform a computation with small input data, such as iterative approximation of values of mathematical functions.

In memory-bound programs, the performance is limited by the speed of memory accesses. This can manifest in common tasks such as binary search or reduce-type operations such as summing.

Both CPU-bound and memory-bound tasks can be easily graded with respect to either processor time or wall-clock time. However, there are classes of tasks where selecting an appropriate grading criterion is more complicated.

IO-bound programs (e.g., external sorting) are limited by the speed of accesses to external memory, such as HDDs or network resources. Due to the inherent instability of access time of external memory, such tasks are hard to measure reliably. Since we need to account for time taken by waiting for IO, CPU time cannot be used for grading this class – an efficient solution will need to work with the external memory in a way that minimizes the IO wait time, which is not included in CPU time (but it is included in wall-clock time).

Exercises in parallel computing (both CPU and GPU based) mostly fall into the CPU and memory-bound categories, but, like in the IO-bound case, we cannot use CPU time to grade them – the CPU time of a multi threaded program is the sum of the CPU times of its threads. Therefore, we cannot use it to measure the speedup gained from parallelization and we are left with wall-clock time. Moreover, we can expect a larger time measurement instability due to the inherent non-determinism of scheduling of multiple threads.

Many assignments are not at all concerned with the performance of the submission and only check its correctness. For those, the evaluation process is similar to unit testing. A de-facto subclass of these assignments are those where the solution is not a computer program in the classical sense – for example, some courses require the students to train and submit a neural network that reaches some level of accuracy on a chosen dataset. Usually, the processing time is not interesting in such assignments, even though it is still necessary to limit it to avoid leaving the evaluation system stuck in an infinite loop.

We will concentrate on two basic groups of workloads – CPU-bound and memory-bound. We expect that the run times of memory-bound tasks will be less stable due to factors such as cache and page misses (these effects are further amplified by virtualization technologies). Apart from that, there are factors that

are detrimental even to the measurement stability of purely CPU-bound tasks – for example, frequency scaling, context switching or sharing of CPU core units when logical cores are being used.

We excluded exercises that are IO-bound or use parallel computing from our analysis. IO-bound tasks are difficult to run in parallel because of shared access to external memory. Also, there are many factors to take into account when running them in a virtualized environment, making their evaluation too complicated for this experiment. Parallel tasks typically require a dedicated machine with a multi-core CPU that should not be used by other measurements. Finally, we do not have to be concerned with the stability of measurements for assignments that are not graded with respect to measured time.

It is also worth noting that being CPU or memory bound is a characteristic of the submitted program and not the assignment. In many tasks, the students can choose the degree of the memory-speed trade-off they want to make (for example, the number of intermediate results stored in a lookup table to avoid recalculation). Also, students might choose to solve problems intended e.g., as CPU-bound with memory-bound programs.

The exercise types we selected for the experiment are:

- `exp`: Approximation of e^x using the $(1 + \frac{x}{n})^n$ formula with x and n as integer parameters that are read from the memory. The calculation itself only uses two integer variables (the parameters) and one float variable (the result). We can expect they will probably stay in CPU registers for most of the execution time. 16384 iterations with pre-generated inputs (`x` and `n`) loaded into memory are performed. This way, the workload tests floating point operations with inputs being read sequentially from the memory.
- `gray2bin`: Conversion of numbers in an in-memory array from Gray code to binary. This workload measures the performance of integer operations while inputs are being read sequentially from the memory.
- `bsearch`: A series of binary searches in a large integer array in the memory. This workload tests random access memory reads, which is a very common memory access scheme in both real-world and synthetic workloads.
- `sort`: Sorting a large integer array in the memory using both the insertion sort and quicksort algorithms. This workload tests a combination of random access and sequential memory reads and writes. This memory access scheme is also common in many real-world and synthetic workloads.

The inputs are generated randomly using the `shuf` command from GNU coreutils. Typically, we generate sets of numbers from a given range, chosen with replacement. According to the documentation, `shuf` chooses the output numbers with equal probabilities (sampling a uniform distribution with replacement).

The input sizes were chosen empirically so that the runtime of a single iteration is between 100 and 500 milliseconds. The main reason for this is that the time values reported by `isolate` are truncated to three decimal numbers and measurements of short workloads would often falsely seem equal to each other due to rounding/truncation of decimals. The iterations should not be too long either, because we measure multiple iterations using multiple isolation technologies, each under multiple execution setups, which totals to a substantial multiplicative fac-

tor on the total runtime. It is also noteworthy that most ReCodEx tests run in tens or hundreds of milliseconds.

The input sizes are as follows:

- `exp`: 65536 random exponents between 0 and 32 with `n=1000` (performed with both `float` and `double` data types)
- `gray2bin`: 1048576 random 32-bit integers
- `bsearch`: 1048576 look-ups in a 65536-item array of 32-bit integers
- `sort/insertion_sort`: 16384 32-bit integers
- `sort/qsort`: 1048576 32-bit integers

2.1.5 Exercise Workload Languages

The core exercises for our experiments shall be implemented in a compiled, low-level language. Such languages should have a relatively small overhead induced by the runtime environment (at least compared to managed languages with features such as garbage collection and JIT compilation).

The most frequently used language in this category as of today is C/C++. Languages such as Fortran, Pascal and Rust could also be considered. We exclude functional languages such as Haskell or OCaml since they operate on a level of abstraction different than that of imperative languages and it could prove difficult to understand how exactly is the code going to be compiled and executed by the CPU. Go is excluded because it features a non-trivial runtime with garbage collection.

For the core exercise types, we selected C as the implementation language. Pascal and Fortran, respectively, are sometimes used in programmer education and scientific computations, but they are not known to the general public as much as C or C++. Rust is a relatively new language that is still evolving rather rapidly. Although it promises memory and concurrency safety thanks to its type system, there are not many ways we could exploit this in our workloads. Also, the adoption is still rather small.

C++ has a multitude of features compared to C, such as type-safe collections and support for namespaces, object oriented programming and template metaprogramming. Its standard library is also much larger. However, the argument that applies to Rust holds here too – our exercises are too trivial to benefit from these features significantly (although templates could make for marginally cleaner code in the `exp` exercise workload). Also, using collections from the standard library could make the measured code harder to reason about in terms of how it will be executed. The final argument for C is that it is still used in many introductory programming courses.

Although a comprehensive study of measurement stability among a large set of programming languages is out of scope of this thesis, it is important to measure with more than one language because computer science programs at universities typically cover more than one.

Admittedly, most courses concerned with the precision of measurements will use low-level languages where we can expect that the measurement stability will be similar to C. However, finding that some class of languages performs poorly in terms of measurement stability would raise a major concern.

We will include a quicksort implementation in Java and Python to see how the stability of measurements is affected by the implementation language. The reason for choosing Java is that it is a language with garbage collection and JIT compilation and it is used in many courses on object-oriented programming. Python on the other hand is a scripting language with many use cases ranging from web development to machine learning. It is also used by introductory programming courses at many universities.

The exercises implemented in Java and Python took much longer per iteration, making the total runtime of the experiment impractical. Therefore, we chose to reduce the input size to 131072 items ($\frac{1}{8}$ of the original size) in order to make their runtime closer to that of the C implementation. This is not a concern since we are not aiming to compare the performance of the implementations anyway.

2.1.6 Measured Data

ReCodEx uses CPU and wall clock time measurements reported by isolate. Therefore, the stability of these values is the most important result of our experiment.

Our workloads are also instrumented manually to measure and report the runtime of the solution (minus the initialization and finalization time of the program) using the `clock_gettime` call. `CLOCK_PROCESS_CPUTIME_ID` is used to measure CPU time and `CLOCK_REALTIME` is used for wall clock time.

This instrumentation is necessary because some isolation technologies cannot provide us with measurements from isolate (in fact, a half of them does not use isolate at all), yet we want to use these technologies in our comparison. Measuring all this data also lets us examine the overhead caused by isolate and any potential discrepancies between the values.

Along with the measurements themselves, we will collect performance data using the `perf` tool that provides access to performance counters in the Linux kernel. We will focus on events that are known to cause unstable run times, such as cache misses and page faults (although our workloads are not very likely to generate a notable amount of page faults). The measurements with `perf` enabled will be run separately to make sure that the profiling does not influence our results. With this data, we will have a better insight into the causes of potential unstable measurements.

The exact counted events are:

- `L1-dcache-loads` – loads from the L1 data cache
- `L1-dcache-misses` – unsuccessful loads from the L1 data cache
- `LLC-stores` – stores to the last level cache (shared by all cores)
- `LLC-store-misses` – memory stores that resulted into a write to the memory (instead of just altering the cache)
- `LLC-loads` – loads from the last level cache
- `LLC-load-misses` – unsuccessful loads from the last level cache that led to a memory load
- `page-faults` – memory loads that led to a page walk

For some workloads, it might be interesting to observe disk-related metrics such as latency. However, the computational workloads we measure are not likely

to be influenced by such factors. Also, this kind of events does not seem to be supported by perf to our best knowledge.

2.2 Hardware and OS Configuration

The measurements will be performed on a Dell PowerEdge M1000e server with two 10-core CPUs. This kind of machine is similar to what could be used for assignment evaluation in a university, for example. The exact specifications are as follows:

- CPU: 2* Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (A total of 20 physical CPUs with HyperThreading) in a NUMA setup
- Memory: 256GB DDR4 (8 DIMMs by 32GB) @2400Mhz

The server runs CentOS 7 with Linux 3.10.0 kernel. CentOS is a freely available Linux distribution functionally compatible with Red Hat Enterprise Linux, a commonly used server operating system. Our operating system is configured according to the recommendations given by the documentation of isolate:

- swap is disabled
- CPU frequency scaling governor is set to `performance`
- kernel address space randomization is disabled
- transparent hugepage support is disabled

Due to the CPU topology, we will measure the following parallel configurations:

- a single process
- 2 processes (each process can use one whole CPU cache)
- 4 processes (2 processes share the last-level cache on each CPU)
- 6 processes (3 processes share the last-level cache on each CPU)
- 8 processes (4 processes share the last-level cache on each CPU)
- 10 processes (5 processes share the last-level cache on each CPU)
- 20 processes (one process per physical CPU core, 10 processes share the last-level cache)
- 40 processes (one process per logical CPU core, 20 processes share the last-level cache)

There are two ways of distributing the measured exercises over CPU cores when measuring with `taskset`. Both are implemented by the `distribute_workers.sh` script. The main idea (shared by both of these approaches) is that the numbers of parallel workers running on each physical CPU should be balanced. The same should apply to logical cores in a physical CPU.

The first approach to workload distribution is illustrated in Figure 2.2. Each workload is assigned to a single core and using two logical cores on the same physical core is avoided as long as possible. Which exact cores are chosen is not important, because the only layer of cache shared by the cores is the last level cache, which is shared by all the cores.

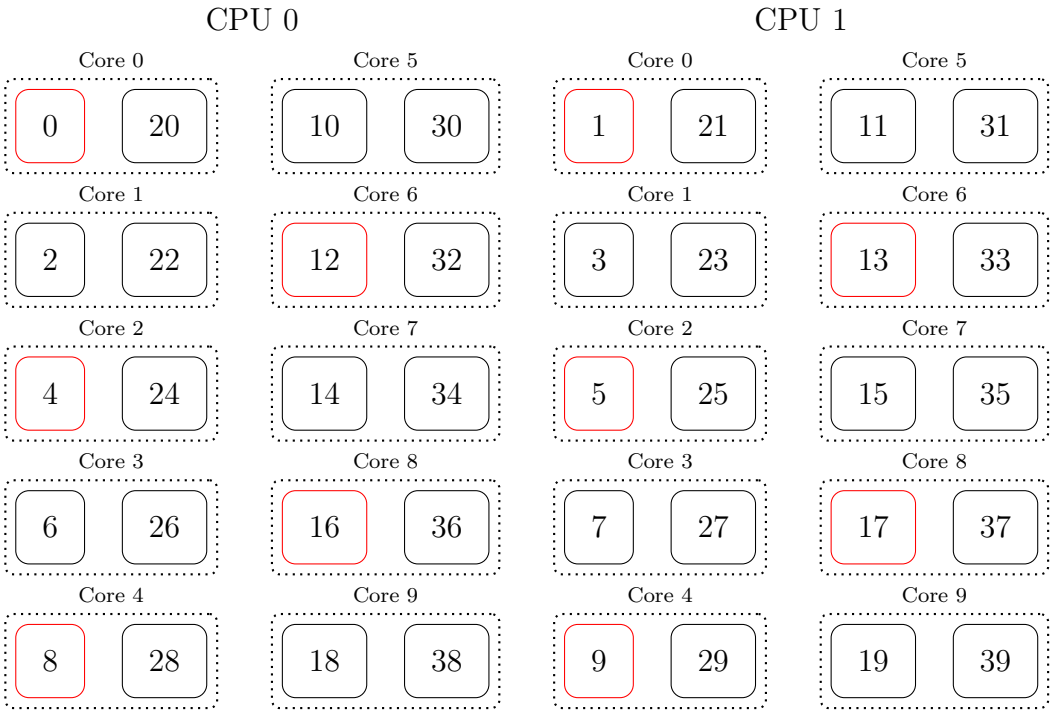

Figure 2.2: Placement of 10 measurements on our CPU cores using the fixed affinity setting policy

The other approach is illustrated in Figure 2.3. It tries to divide the CPU cores into equally-sized sets where logical cores that belong to the same physical core always belong to the same set.

The `distribute_workers.sh` script might require adjustments if we try to replicate this experiment on other CPUs with different topologies - in other words, it does not attempt to cover all possible CPU configurations.

2.3 Analysis of Measurement Conditions

Considering the objective of our experiment, we had to ensure that the results of our measurements are stable in ideal conditions (only one process at a time being measured on the bare metal without isolation) in the first place. Otherwise, the comparison with results in less than ideal conditions would be much more difficult. In other words, we are going to make sure that conditions exist under which the exercises we chose yield stable results.

2.3.1 Dependence of Result Variance on Input

Being able to use randomly generated inputs in the measurements of workloads is very useful – we can demonstrate that the outcome of our measurements was not influenced by carefully choosing inputs that yield the desired results. This can be done by simply regenerating the input data and seeing if we get the same outcome. However, this only holds when the generated inputs are large enough so that the measurements take the same amount of time on every repetition.

To see if the input sizes we chose are sufficient, we measured the execution

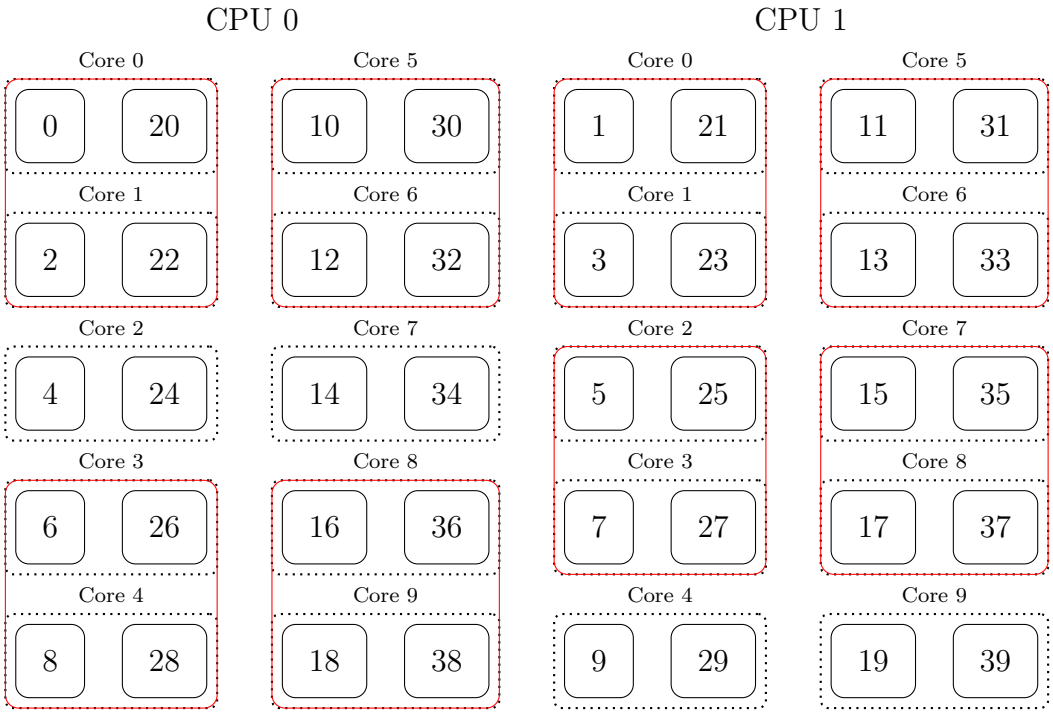

Figure 2.3: Placement of 8 measurements on our CPU cores using the multi-core affinity setting policy

time of each workload (100 iterations) on 300 randomly generated input files and calculated the mean and standard deviation of the measurements for each of the inputs. As we see in Figure 2.4, the mean execution time almost does not vary – even the outliers are within milliseconds from the median of the means. However, as shown in the same figure, the range of standard deviations is rather large, reaching up to 11ms. Upon closer inspection, we found that this is due to a small number of outliers. We conclude that the input data has a negligible effect on the execution time, even though there is a handful of inputs for the `qsort`, `bsearch` and `gray2bin` exercise types on which the time measurements exhibit a notably higher standard deviation.

2.3.2 Behavior of Repeated Measurements

In computer performance evaluation, it is common to let the benchmark warm up by performing a handful of iterations without measuring them. This way, the measurements are not influenced by initialization of the runtime environment or population of caches, for example.

We expect that warming up will not occur in our experiment because each iteration runs in a separate process and actual submissions are different binaries. However, factors that could cause this phenomenon in our case do exist. For example, if the submission read a very large input file, it would have to wait for it to be read from the disk, but subsequent submissions could probably get it from the disk cache. Also, many successive submissions of a short program could vary in their runtime thanks to CPU frequency scaling. Therefore, it is still necessary to verify whether or not warming up occurs.

As seen in Figure 2.5, there is almost always an outlier in one of the first

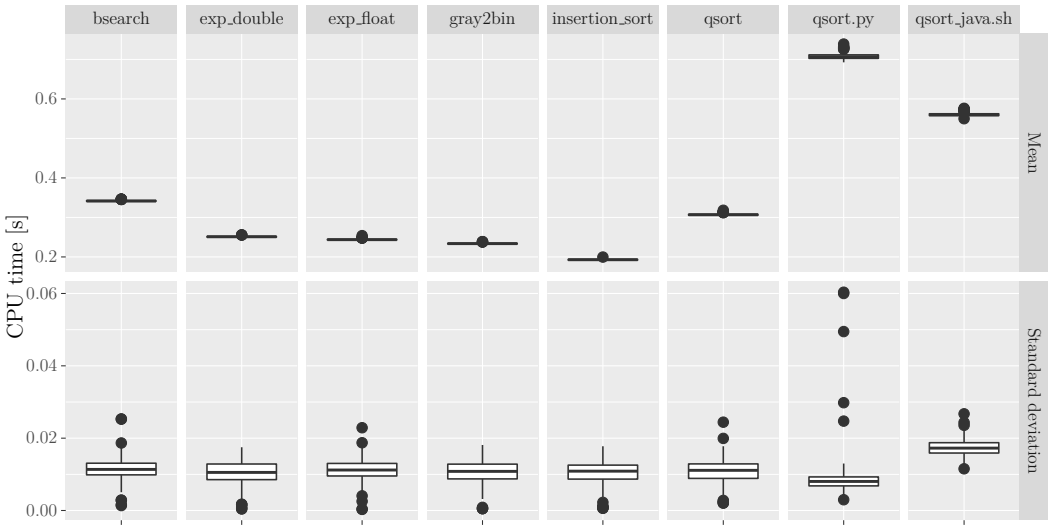

Figure 2.4: A box plot of the iteration means and standard deviations of CPU time for each workload

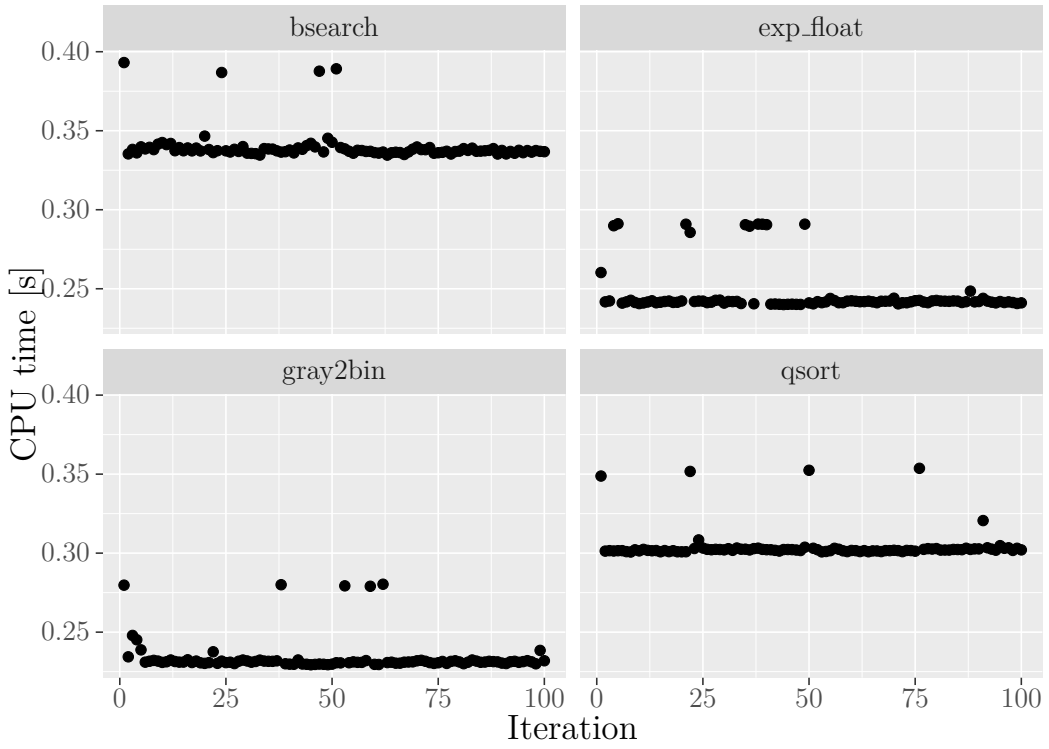

Figure 2.5: A scatter plot of CPU times for selected exercise types with no isolation and a single measurement worker running

iterations. A handful of outliers with similar execution time can however be observed throughout every iteration of all the benchmarks. This situation does not seem like the usual case of warming up where the first results are regularly higher than the later ones and where the execution time does not rise in subsequent iterations.

Although it is possible that 100 measurements is not enough to detect such a warm-up period, it seems improbable. Therefore, we can conclude that warming up is not an important factor in our measurements, even though it happens on a small scale.

If the opposite was true, we would have to change the way ReCodEx measures submissions – if a student submitted the same program in a quick succession, they could get a better score for the later solution.

2.4 Result Analysis

2.4.1 Evaluation of Isolate Measurements

As mentioned in Section 2.1.6, our measurements that run in isolate yield four values: the CPU and wall clock times as measured by isolate and by the program itself. It is safe to assume that there will be some discrepancies between the results from isolate and from the program – isolate also takes into account the time it takes to load the binary and start the process, as opposed to the instrumented measurements start when everything is ready. However, since we only observe the values reported by isolate in ReCodEx, this does not concern us as long as the error is deterministic (i.e., it stays the same in repeated measurements). Additionally, if the error was deterministic, but larger than the runtime of the program itself, it would be more difficult to recognize inefficient solutions.

To examine this, we took the results of measurements with the `parallel-homogeneous` execution setup type and made a correlation plot of times reported by the program itself and by isolate for each exercise type and both CPU and wall-clock time measurements. There is a problem with the interpretation of these plots – execution setups with many parallel workers yield more observations in total. Therefore, the results from highly parallelized execution setups are more prominent in the correlation plots. To alleviate this, we plotted a random sample from the observations using the inverse of the number of parallel workers as a weight for each observation (e.g., an observation from a setup with 40 workers has a weight of $\frac{1}{40}$ and is therefore less likely to be selected than an observation from a setup with 10 workers which has a weight of $\frac{1}{10}$).

As depicted in Figure 2.6, the error is rather small and stable for the CPU times. This result is not surprising – starting a process is generally not a CPU-intensive task. Measurements of the quicksort workload in Java are an exception – the times measured by isolate were twice as long in almost every iteration. Also, the results seem less stable when the execution time is longer. We can attribute this difference to the work required to launch the JVM. Still, the measurements for all the other exercise types seem fairly reliable.

On the other hand, we found that the wall-clock time error tends to vary a lot (Figure 2.7). To find out if there is a correlation between the exercise type,

isolation technology or system load level and the error rate, we calculated the mean and standard deviation of the difference between the times measured by the program and by isolate for each iteration and grouped them by exercise type, isolation, execution setup type and load level. We also normalized the standard deviation by dividing it by the mean of the runtime to obtain a relative error measure. The results can be seen in Attachment A.

We found that the instability in wall-clock time measurements is most prominent when a high number of parallel workers is involved (20-40) – the relative error goes as high as 196%. Smaller values of the relative error (over 5%) start to manifest with as little as six parallel workers. This means that there is a large offset between the wall-clock time measured by isolate and the value measured by the programs themselves. Also, this offset tends to vary a lot in many cases.

We found no obvious link between the value of the relative wall-clock time error and the isolation technology in use – both docker+isolate and isolate on its own tend to have largely varying measurement errors. VirtualBox might seem to be more stable at first glance because we are missing data for larger amounts of parallel workers.

The instability of the error is possibly caused by the overhead of starting new processes. This overhead grows larger when we need to start many processes at once and both the file system and memory get stressed.

2.4.2 Method of Comparison

In the following sections, we will need to compare groups of measurements made under different conditions (e.g., with CPU affinity settings or under varying degrees of system load). We cannot make assumptions about the distribution of the data, which disqualifies well-known tests such as the pairwise t-test, which requires normally distributed data.

Our approach is to compare confidence intervals of characteristics (such as the mean or standard deviation) of the different groups. When the confidence intervals of a characteristic do not overlap for a pair of groups, we can conclude that the characteristic differs for the groups. When one of the intervals engulfs the other, the characteristic is probably the same. When the intervals overlap, we cannot conclude anything.

We obtain the confidence interval using the Bootstrap method[23], which is a resampling method that does not rely on any particular distribution of the data. The core idea is that we take random samples of our measurements repeatedly (1000 times in our case), calculating the statistic whose confidence interval we are trying to estimate in each iteration. This way, we get a set of observations of our statistic. Then, we select the 0.05-th and 0.95-th percentile to receive a 0.95 confidence interval. The implementation we use is provided by the `boot` package for the R statistic toolkit.

2.4.3 Isolation and Measurement Stability

To visualize the effects of isolation technologies, we made scatter plots of CPU time (we chose not to examine the wall-clock time because, according to Section 2.4.1, it seems that isolate cannot measure it reliably) measurements for each

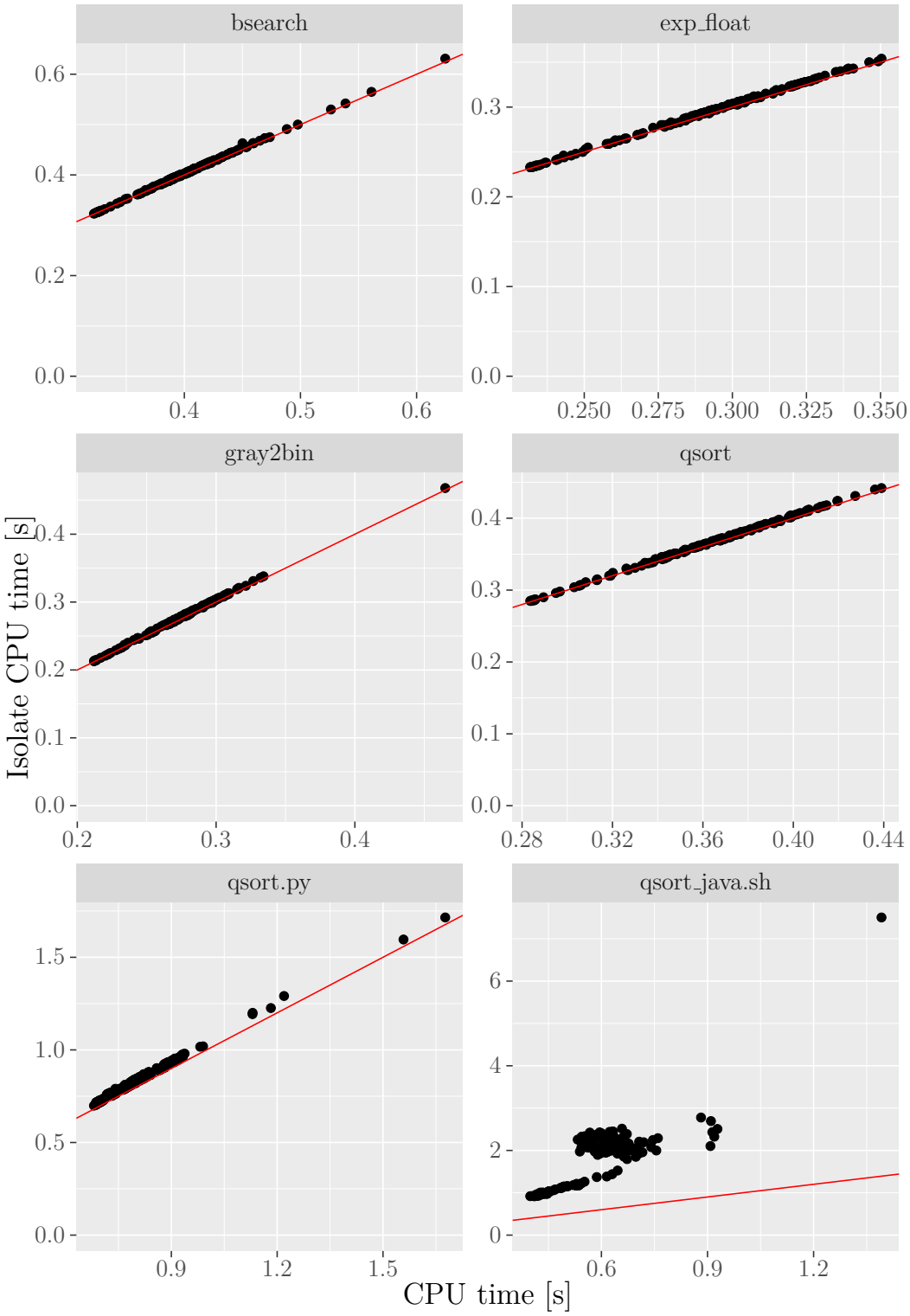

Figure 2.6: A correlation plot of CPU time reported by the measured program and by isolate, with y=x as a reference line

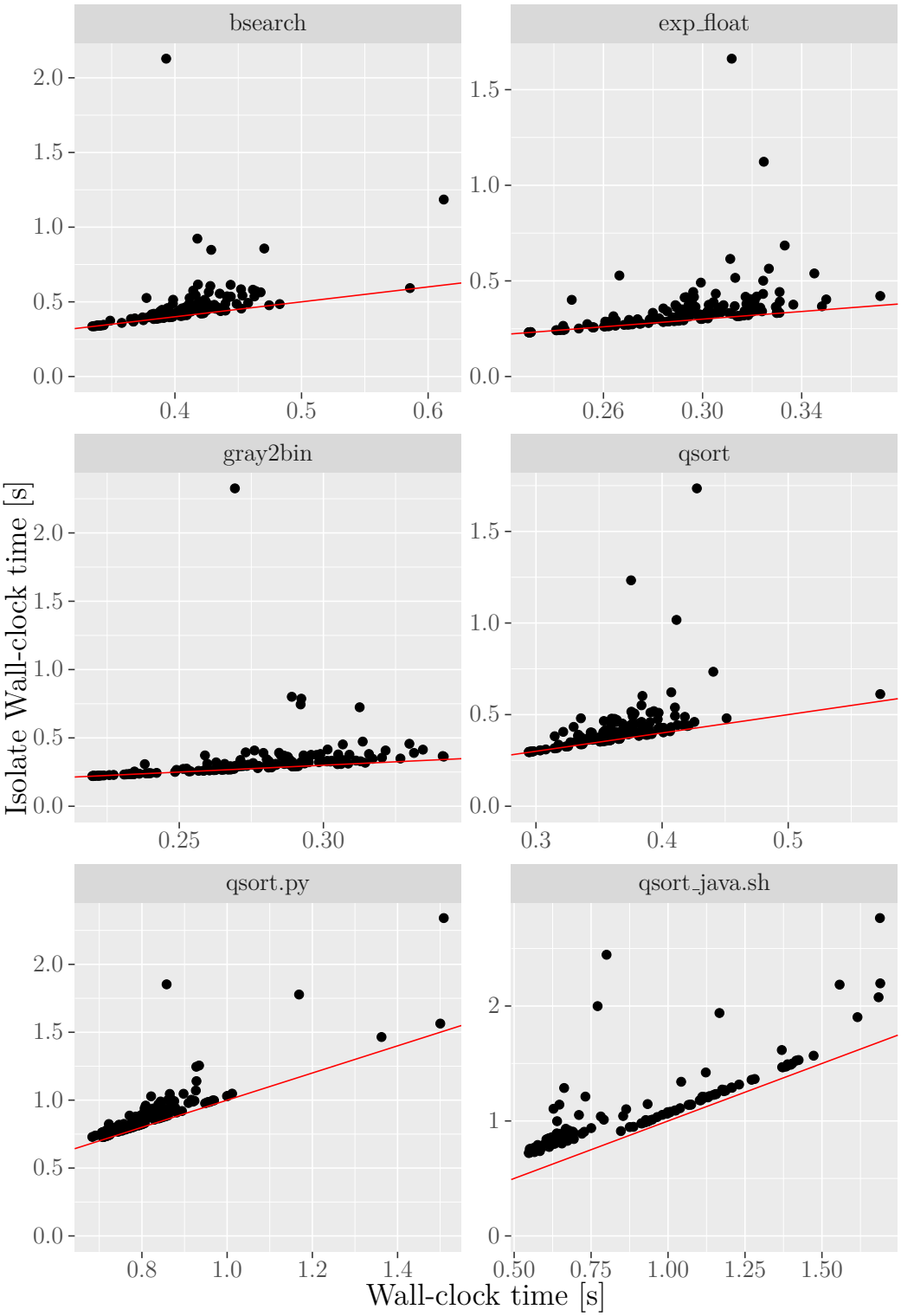

Figure 2.7: A correlation plot of wall-clock time reported by the measured program and by isolate, with y=x as a reference line

system load level, exercise type and isolation technology so that the plots for each isolation technology are side by side for every exercise type. Because most of the measurements were made in parallel, we colorized the measurements from one chosen worker process so that we could get an idea about how the measurements differ between the parallel workers. The plots revealed a handful of possible trends in the measured data. A selection from these plots can be seen in Figure 2.8.

The most prominent trend is that the measured values are centered around different means under different isolation technologies for some exercise types.

For example, an iteration of the `bsearch` and `exp_float` workloads tends to take about 25-50ms more on the bare metal or in Docker than in VirtualBox. This trend is most apparent with four or more workers running in parallel.

Also, there is a cca. 50ms difference between the `I` and `D` (and also between `I` and `B`) setups on a single worker, which is rather strange because both of these technologies use the same kernel facilities to achieve isolation.

To examine the first observation in a more formal way, we compared 0.95 confidence intervals of the mean and standard deviation for the `bare`, `docker-bare` and `vbox-bare` isolation setups. The comparison was performed separately for each execution setup type and system load level and exercise type. As shown by Figure 2.9, the comparison of the means confirmed our observation – measurements in VirtualBox often yield lower times than in Docker and on the bare metal. Measurements in Docker also yield lower times than on the bare metal most of the time. The comparison of the standard deviations suggests that measurements in VirtualBox are more stable than those on the bare metal and in Docker and that there are no notable differences in stability between Docker and the bare metal.

We assessed the effect of adding Isolate to a setup in a similar way – we grouped the measurements by execution setup type and system load level, isolation technology and exercise type and compared confidence intervals of the mean and standard deviation among groups of measurements with and without Isolate. The results of this comparison (Figure 2.10) show that measurements with Isolate are generally slower (exhibit a higher mean) on the bare metal and with Docker. In VirtualBox, adding Isolate does not seem to influence the mean in any obvious way.

However, we found that the addition of Isolate reduces the standard deviation of measurement in many cases. This trend is most prevalent on the bare metal, but it does also manifest in Docker. In VirtualBox, it is safer to conclude that the standard deviation remains the same.

As a side note, in VirtualBox, the wall-clock time measurements of some workloads such as `qsort.py` tend to have larger outliers (such as 12 seconds when the median is around 6 seconds). However, this phenomenon only manifests in a few cases and we decided not to study it further.

2.4.4 Validation of Parallel Worker Results

For the homogeneous parallel setup, we needed to make sure that our measurements did actually run in parallel. We calculated the total runtime for all the measurements (the difference between the timestamps of the first and last mea-

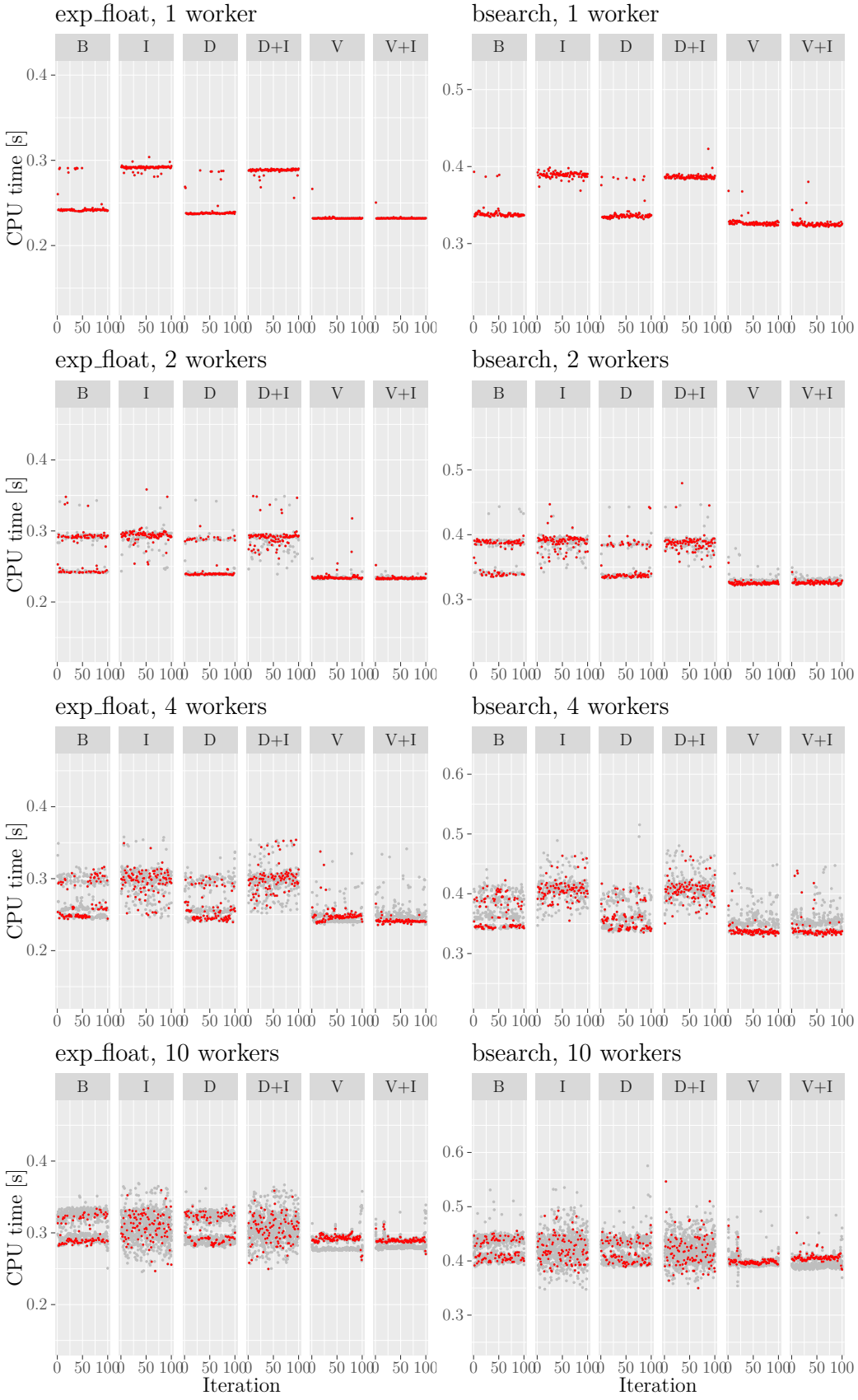

Figure 2.8: A scatter plot of time measurements grouped by isolation for chosen setups and exercise types with results from a single worker highlighted in red

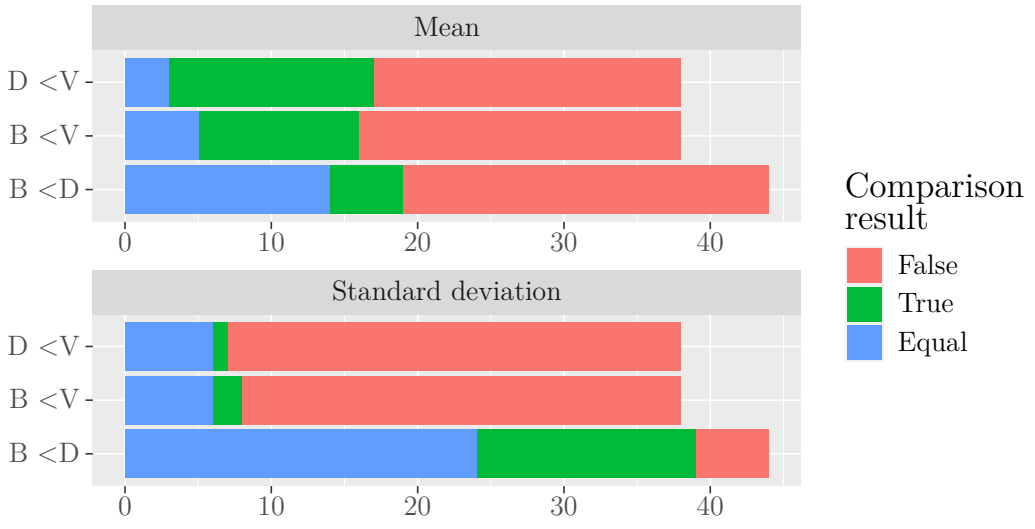

Figure 2.9: Results of comparisons of confidence intervals of means and standard deviations among various measurement groups, divided by isolation technology

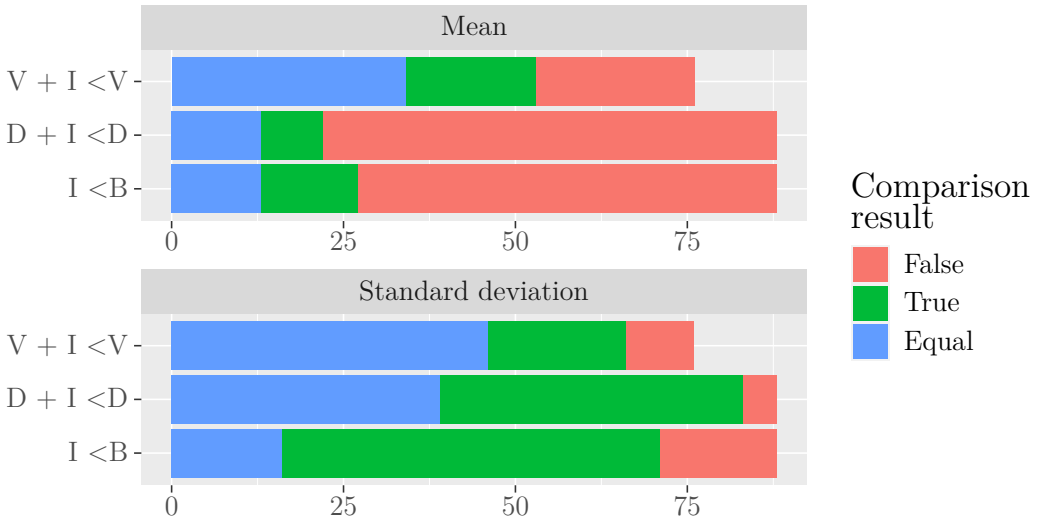

Figure 2.10: Results of comparisons of confidence intervals of means and standard deviations among various measurement groups, with and without isolate

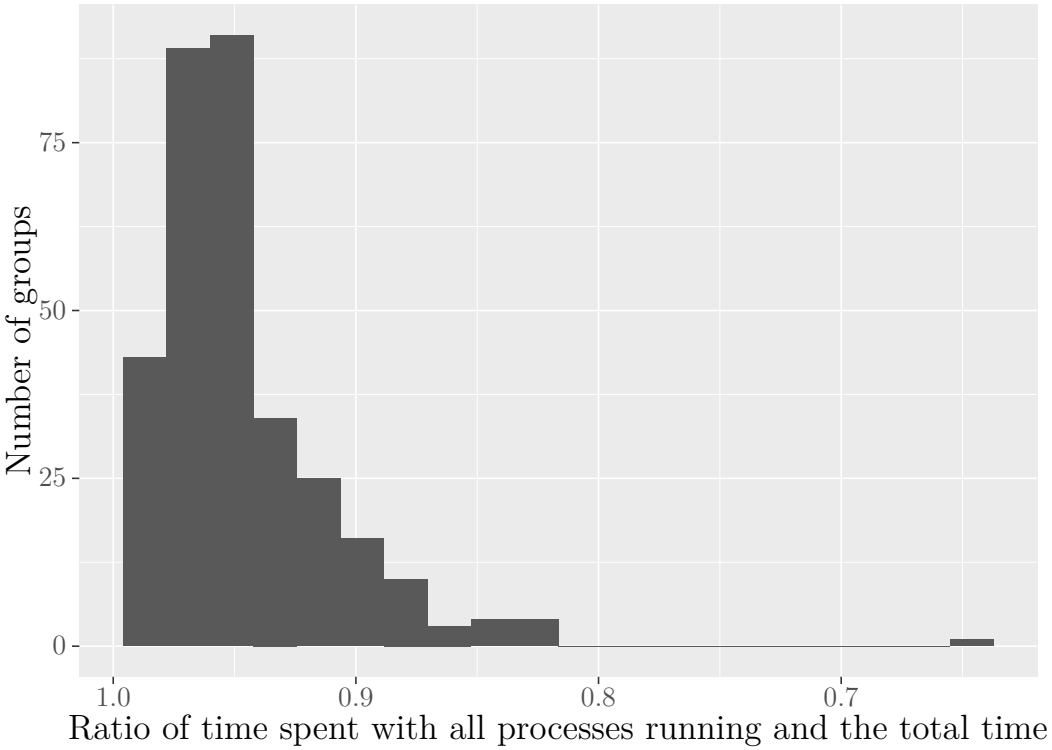

Figure 2.11: A histogram of parallel run ratios

sured results, regardless of the worker) and the time when all the worker processes ran in parallel (the difference between the timestamps of the last received result of a first iteration and the first received result of a last iteration). We then inspected the ratio of these two numbers.

As depicted in Figure 2.11, this ratio is 68% in one case and higher than 90% most of the time. This should guarantee that the homogeneous parallel measurements are not in fact a sequence of sequential workloads and their results are similar to actual parallel measurements.

2.4.5 Comparing Parallel Worker Results

When examining how raising the system load affects the stability of measurements, we discovered several interesting trends.

First, the execution times seem to be higher under higher levels of system load for all exercise types. For example, in Figure 2.12, we can see that a `bsearch` iteration normally takes about 0.35 seconds on the bare metal with a single process and 0.4-0.45s when 10 processes are running in parallel. It is worth mentioning that the difference seems much smaller for `exp_float`. This could be due to the parallel workers competing for the last-level cache and memory controller, which would affect memory-bound tasks more than it would affect CPU-bound tasks.

Second, the measurements seem to be notably stable with a single worker on the bare metal (`B`) and in Docker (`D`). This stability, however, decays quickly with as little as two workers measuring in parallel. In `isolate` (`I`) on the bare metal, in Docker with `isolate` (`D+I`) and in VirtualBox (`V` and `V+I`), the stability of measurements seems similar in the cases with one and two parallel workers.

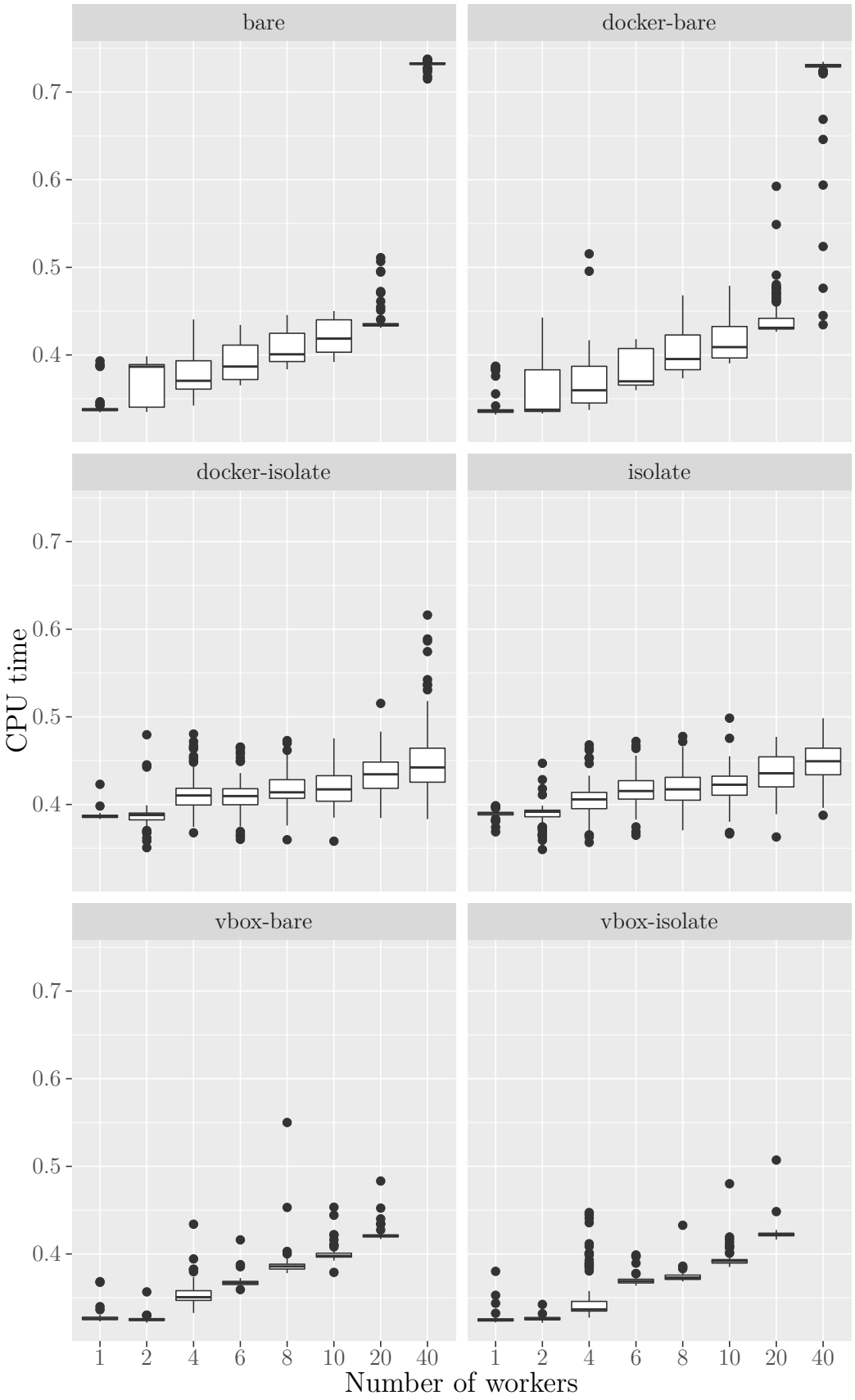

Figure 2.12: Box plots of CPU time measurements for the bsearch workload with an increasing number of parallel workers (divided by isolation technology)

A noticeable decay appears with four workers. These trends are illustrated by Figure 2.8.

Third, the instability related to a higher degree of parallelism seems more prominent in measurements of memory-bound workloads than in those of CPU-bound workloads. This could however be a coincidence since the CPU-bound workloads take less time per iteration than the memory-bound ones.

A conclusion follows from our observations: we cannot allow a number of parallel workers that is so high that any of these effects manifests. Otherwise, we risk that our measurements will become too unstable. In the case of our hardware, it seems that running as little as four workers in parallel might lead to a measurable instability.

2.4.6 Evaluation of Performance Metrics

In this section, we try to explain the changes in results of our time measurements caused by using `isolate` and increasing the system load using the data gathered by `perf` (the exact list of counted events can be found in Section 2.1.6). With the exception of `page-faults`, the events form pairs of the number of misses and the total number of accesses for a type of cache. Therefore, it is natural to also examine the miss ratio for these counters, which is depicted in Figure 2.13.

The miss ratio for L1 data cache loads seems close to zero for every exercise type except `bsearch`. However, the absolute number of misses is substantial (in the order of millions of events per iteration). The miss ratio is also higher when using `isolate`, although it is still in the order of tenths of a percent. No change in the number of misses was observed with increasing system load. This shows that the values in L1 cache are used very frequently and it performs well in all cases.

For stores in the last level cache, the miss ratio seems larger with `isolate` than on the bare metal. However, it does not increase with the system load as much. The ratio is mostly less than 0.05% on the bare metal and close to 0.1% with `isolate`. This might be a part of the explanation for `isolate` measurements being slightly less unstable with increasing number of parallel workers than measurements on the bare metal.

We observed an unexpectedly small last level cache load miss ratio in `bsearch` (around 0.5%), when compared to the other exercise types (up to 60%). This could indicate that our binary search workload utilizes the last-level cache more than the other workloads, which is plausible – the other exercise types seem to work with the L1 data cache more efficiently and might not need to use the last level cache as much. We could not find any interesting trend in the data related to neither using `isolate` nor increasing the system load.

To see if there is a relationship between the results of measurements of performance metrics and the measurements of CPU time, we calculated the standard (Pearson) correlation coefficient and also the Spearman coefficient of each of the performance metrics and the CPU time for every tested exercise type. The Spearman coefficient should help in pointing out nonlinear relationships between the variables.

The correlation coefficients are listed in Table 2.1. It seems that no value from our selection influences the CPU time. The only result that does not clearly

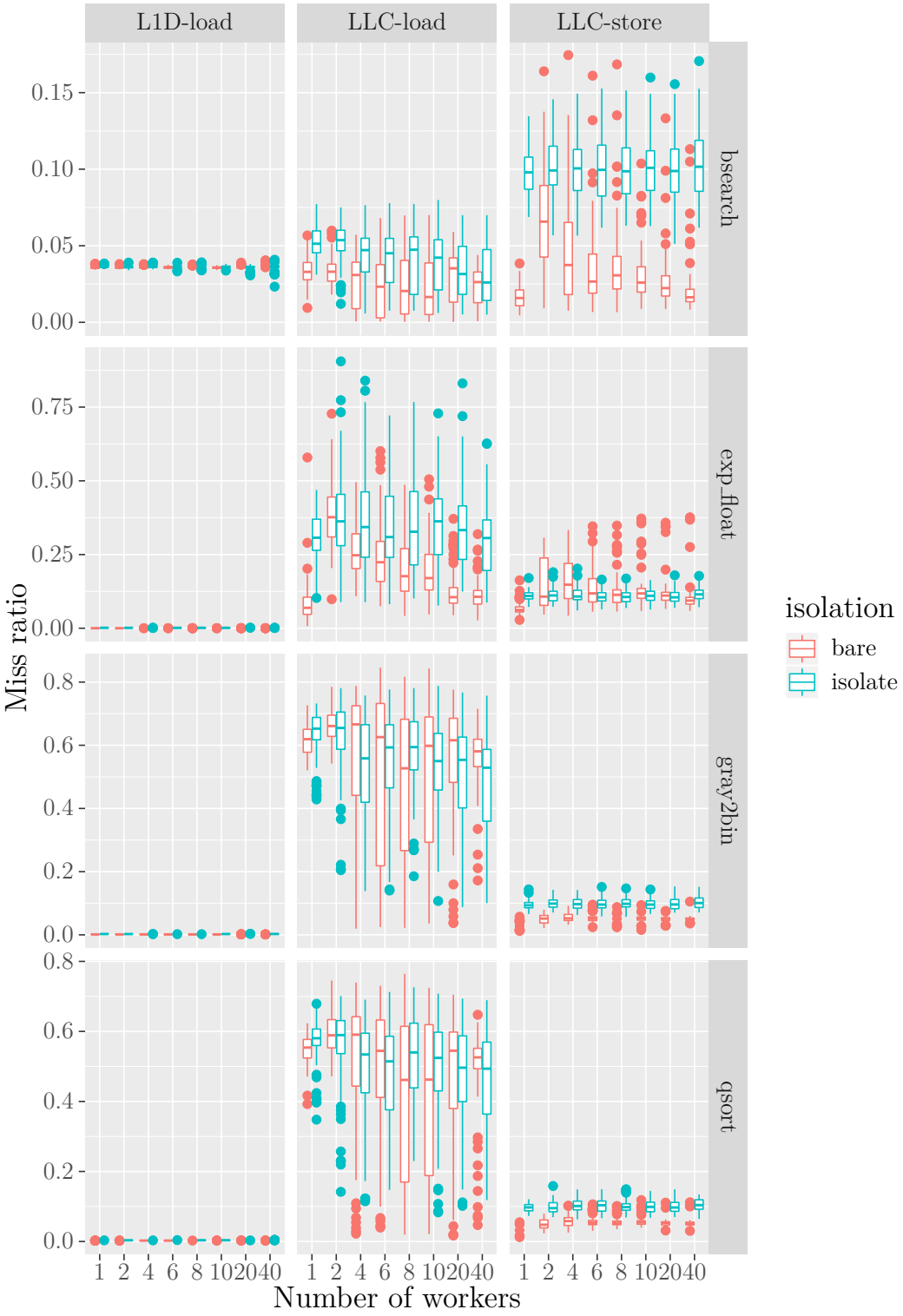

Figure 2.13: The ratios of the number of miss events and number of load/store events for the L1 data cache (L1D) and the last-level cache (LLC), shown as box plots divided by isolation technology and number of parallel workers

suggest that the values are unrelated is that the standard (Pearson) correlation of the number of L1 cache misses and the CPU time is `0.417` for the `bsearch` workload. However, this is not nearly enough evidence for any conclusion. Also, even though L1 data cache misses occur more during the `bsearch` workload than during other workloads, their frequency does not seem to be greatly influenced neither by the number of parallel workers nor by using `isolate`.

Table 2.1: Pearson (standard) and Spearman correlation of CPU time and selected performance metrics

Metric	Workload	Median	Pearson	Spearman
L1_dcache_misses	exp_float	270984.5	-0.009	0.103
LLC_store_misses	exp_float	14635.5	-0.016	0.016
LLC_load_misses	exp_float	7047.5	-0.013	-0.003
page_faults	exp_float	396.5	-0.034	-0.079
L1_dcache_misses	bsearch	15166744.0	0.417	0.254
LLC_store_misses	bsearch	16644.0	-0.070	0.038
LLC_load_misses	bsearch	94619.5	-0.111	-0.108
page_faults	bsearch	388.5	-0.082	0.015
L1_dcache_misses	gray2bin	826283.5	-0.002	0.100
LLC_store_misses	gray2bin	17990.0	-0.125	-0.003
LLC_load_misses	gray2bin	123149.5	-0.076	-0.088
page_faults	gray2bin	1348.5	-0.143	-0.088
L1_dcache_misses	qsort	1570664.0	0.083	0.148
LLC_store_misses	qsort	19243.5	-0.099	0.037
LLC_load_misses	qsort	124440.0	-0.080	-0.086
page_faults	qsort	1348.5	-0.117	-0.041

The last metric left to examine is the number of page faults (depicted in Figure 2.14). We can see that it does not increase with the number of parallel workers, which is not a surprising result. Apart from this, it is evident that using `isolate` results in cca. 300 page faults, regardless of the workload type. This is likely the cost for loading the `isolate` binary, which is also measured by `perf`.

The result of our analysis is that we could not find any explanation of the phenomena observed in the previous section based on the data obtained from `perf`. If the experiment was to be repeated, a more careful selection of observed events would be required.

2.4.7 The Effects of Explicit Affinity Settings

Process schedulers in modern operating systems for multi-CPU systems are complex software that relies on sophisticated algorithms. Trying to bypass the scheduler by pinning worker processes to CPUs with taskset would not be generally recommended.

However, schedulers are typically concerned about objectives such as throughput, latency of IO-bound tasks and fairness (prevention of starvation)[24], and measurement stability is not an important consideration when they assign processes to CPU cores, so it makes sense to examine the effects of setting the CPU

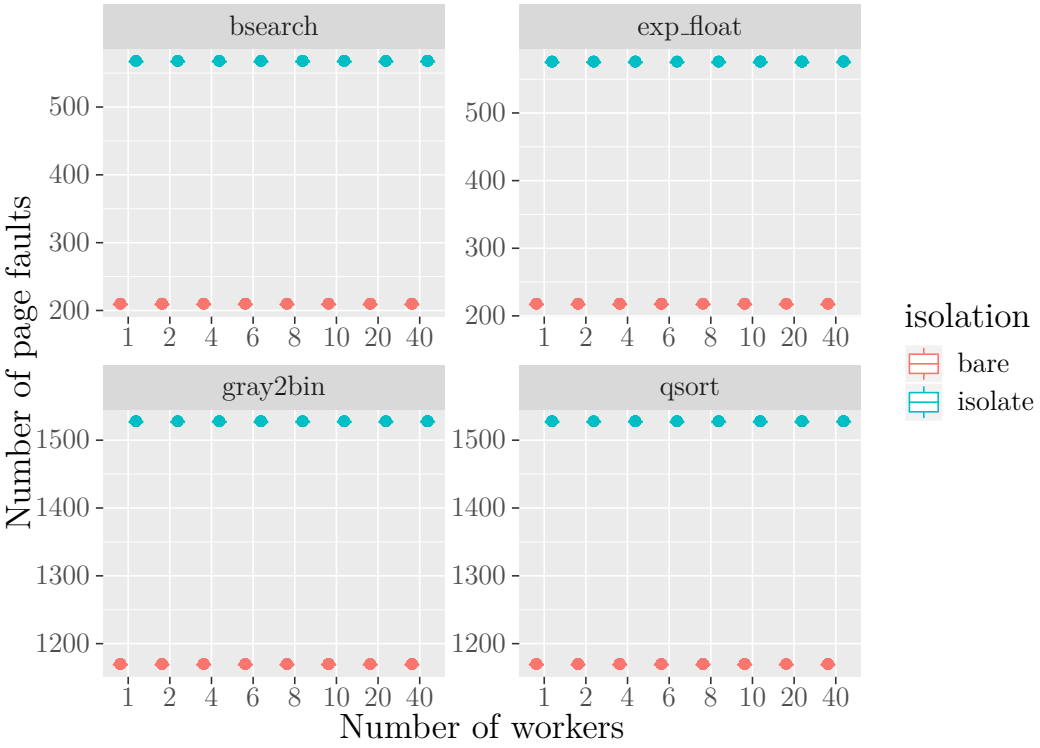

Figure 2.14: Box plots of the number of page faults, divided by number of parallel workers, workload type and isolation technology

affinity explicitly.

We ran our experiment using three different ways of setting the affinity – using `numactl`, using `taskset` with a single core and using `taskset` with a fixed subset of the available cores (this is elaborated on in Section 2.2).

To get an insight into the effects of CPU affinity settings, we compared the means and standard deviations of results grouped by exercise type, isolation and execution setup, with and without affinity settings. We performed the comparison using confidence intervals obtained from a bootstrap procedure (the same way as we used in previous sections).

In Figure 2.15, we can see that the measurements performed with `numactl` have the same mean and standard deviation as their counterpart without affinity settings in most cases. This seems to confirm the assumption that running processes on the memory node that belongs to the executing CPU is the default behavior. Therefore, using `numactl` would be redundant.

Fixating a worker process to a single core with `taskset` seems to make the measurements less stable in terms of standard deviation in two thirds of the compared groups. It is clear that the mean was influenced by this setting in some way, but it is difficult to summarize the difference – the number of groups where it got smaller is very similar to the number of groups where it got higher. Since the comparison did not yield any positive results, we will not consider the single-core `taskset` setup any further.

Finally, the multi-core way of using `taskset` seems to improve both the mean and the standard deviation in more than three quarters of the compared groups. This seems like a notable breakthrough. Upon closer inspection, we found that

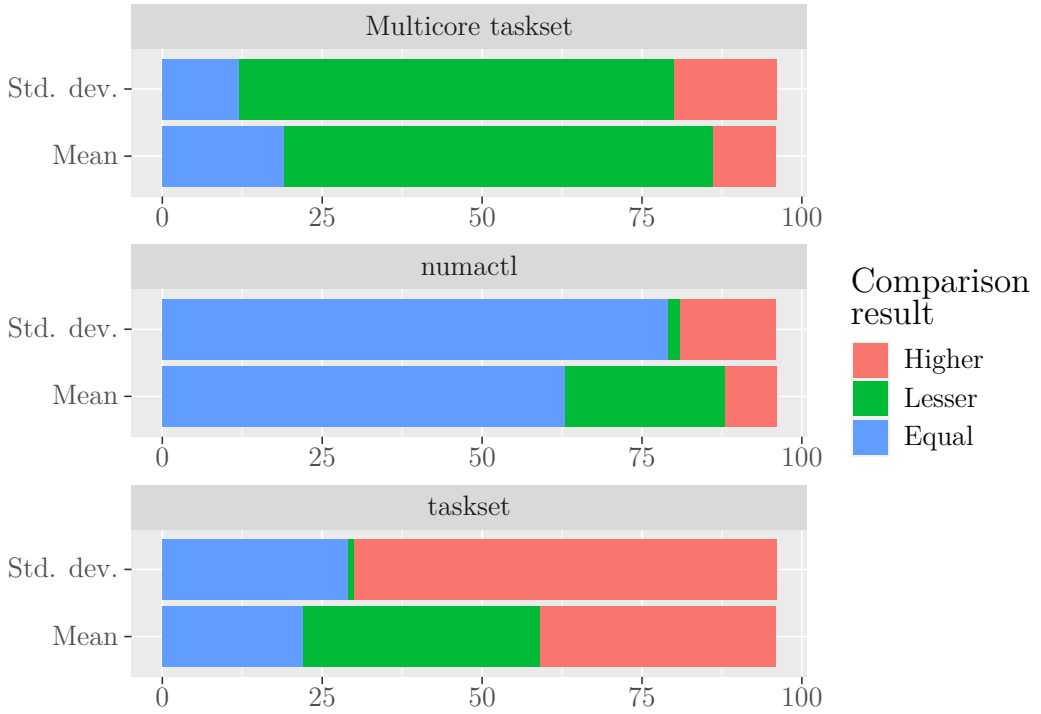

Figure 2.15: A plot showing the results of a comparison between mean and standard deviation values for measurements with and without explicit affinity settings, for different ways of setting the affinity

the results seem much more stable without any isolation technology (as shown by Figure 2.16). However, it is worth noting that the mean execution time still rises with the increasing setup size. For example, `bsearch` takes roughly 0.35ms on two workers and about 0.425ms on eight workers (20% more).

Furthermore, it can be seen in Figure 2.17 that using multi-core `taskset` does not cause any improvement when we use `isolate` for process isolation. The results of measurements in Docker without `isolate` look similar to those of measurements with no isolation at all. Docker with `isolate` performs similarly to `isolate`.

From these observations, we can conclude that using multi-core taskset could help stabilize measurements on the bare metal or in Docker in case they were run in batches with a fixed number of workers. However, setting the CPU (or NUMA) affinity does not bring any improvement in the case of ReCodEx, where the number of active workers varies in time and where isolation is critical.

2.4.8 The Effects of Disabling Logical Cores

Disabling logical cores (HyperThreading) might improve the stability of measurements, since it stands to reason that processes that run on different logical cores in a single physical core influence each other more than processes on separate physical cores.

As a side note, disabling HyperThreading can also help increase the security of a server, since many exploits have surfaced recently that use this technology.

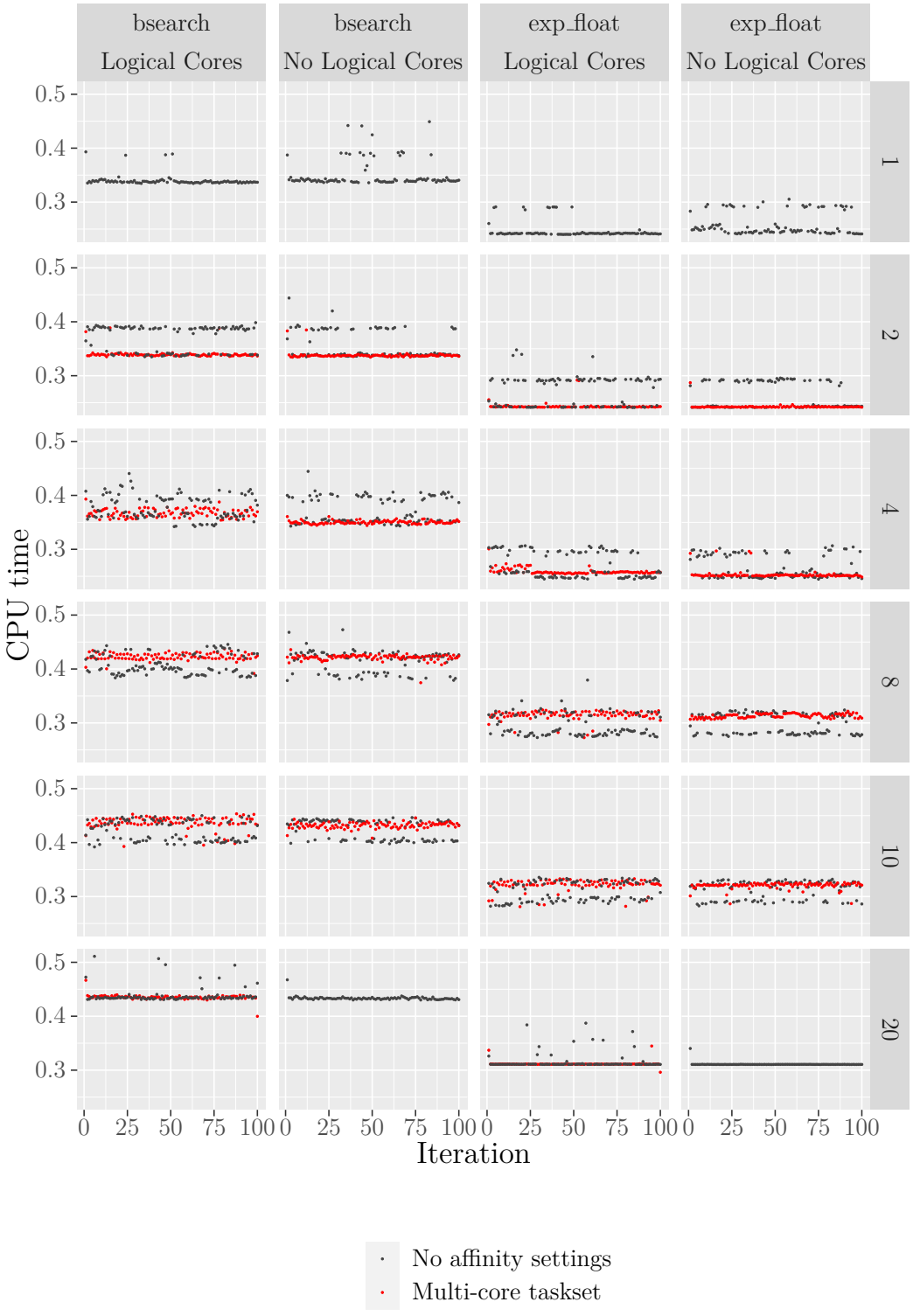

Figure 2.16: A scatter plot of measured CPU times by iteration for increasing setup sizes (no isolation technology)

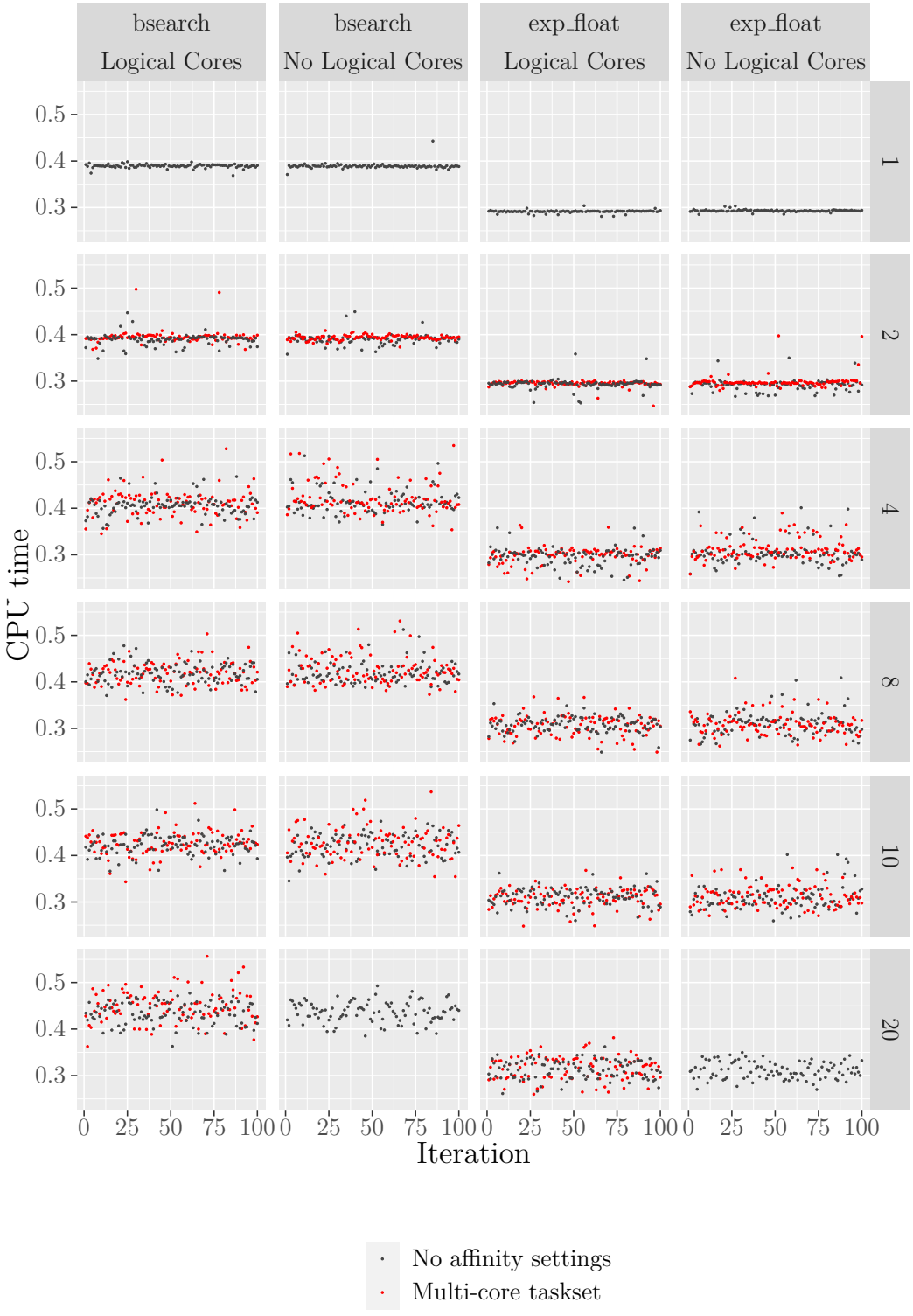

Figure 2.17: A scatter plot of measured CPU times by iteration for increasing setup sizes (using isolate for process isolation)

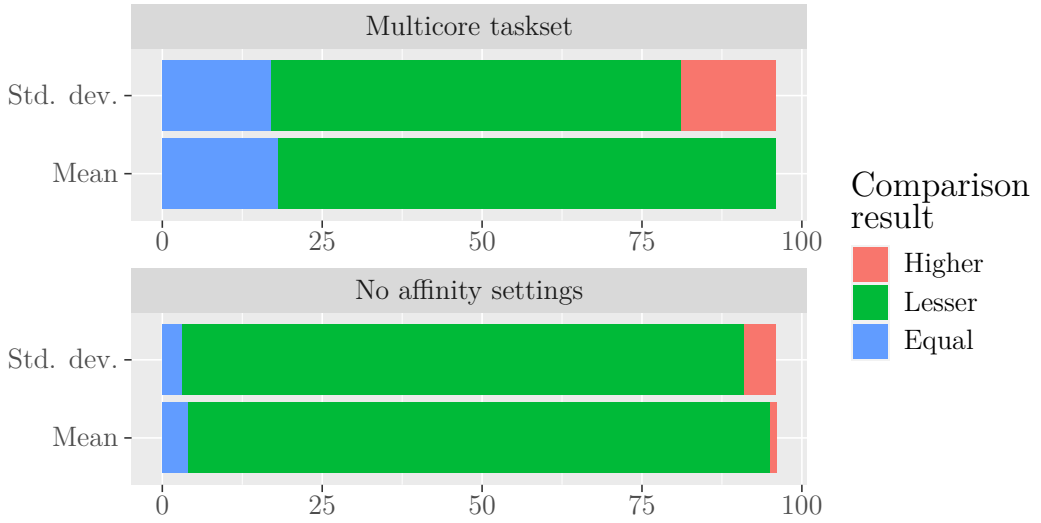

Figure 2.18: A plot showing the results of a comparison between mean and standard deviation values for measurements with and without logical cores enabled. One group of comparisons was made with no explicit affinity setting and the other was made with the multi-core taskset policy.

We do not expect students to submit functional HyperThreading exploits into ReCodEx, but it might be a concern if we were building a public programmer education platform.

To verify our hypothesis about stability, we repeated the measurements on the same machine with the HyperThreading feature disabled (using a startup configuration option). The results for selected workloads on the bare metal and with `isolate` can be seen in Figure 2.16 and Figure 2.17, respectively. It seems that disabling logical cores increases the stability for bare metal measurements in some workloads and that the multi-core taskset setting further improves the situation. This trend is not as prominent when `isolate` is used. In fact, the scatter plot of measurements without logical cores in Figure 2.17 looks almost indistinguishable from that with logical cores enabled.

A comparison of confidence intervals of the means and standard deviations for all groups of measurements by workload type, execution setup and isolation technique (as depicted in Figure 2.18) shows that with no affinity settings, the measurements with logical cores disabled have a lesser mean and standard deviation that those with logical cores enabled in almost all cases.

The results are not as conclusive when we compare measurements performed with the multi-core taskset setting (again, we compare measurements with logical cores disabled to those that use logical cores). While the mean is decreased in the majority cases and increased in none of them, the standard deviation increases in about 20% of the cases.

In summary, it seems that disabling logical cores improves the overall stability of measurements, but the improvement is not as large when `isolate` is used.

2.5 Conclusion

The experiment provided us with evidence that isolate has an effect on the stability of CPU time measurements. Measurements with isolate exhibit a higher mean, but a reduced standard deviation. The exact cause of this remains to be researched, along with the curious case of VirtualBox, where this effect does not seem to be present. Also, the measurements in VirtualBox seem to be faster and more stable than those on the bare metal and in Docker.

Also, we found that measuring many submissions at once impacts the stability of measurements. On a system with two 10-core CPUs, a notable decrease in stability appeared with as little as 4 parallel workers. However, the requirements for measurement stability vary for different assignments and decisions about worker deployment should be based on test measurement data.

The fact that measurements on the same machine interfere with each other has one more consequence – using common virtualized platforms for evaluation is not an option when stable measurements are required, since we have no control of neither the hardware performing the measurements nor other programs being executed in parallel.

Our experiment also yielded three smaller results. First, the wall-clock time measured by isolate tends to be unstable and should not be trusted when high precision measurements are required. Of course, this phenomenon should be researched further, possibly with newer versions of the kernel.

Second, setting the CPU affinity explicitly does not generally yield any improvements to the overall measurement stability, even though the multi-core affinity setting policy seems to improve the stability for batch measurements on the bare metal. The results with `isolate` are less conclusive

Third, disabling logical cores also seems to improve measurement stability on the bare metal when the number of parallel workers is constant. Same as in the case of multi-core affinity setting, this phenomenon is less evident when `isolate` is used, and therefore less likely to bring any practical improvements.

2.5.1 Discussion

It seems that taking advantage of servers with multiple CPU cores for assignment evaluation is difficult due to the instability introduced by parallel measurements. Here, we present a handful of proposals that could improve the situation, and that could serve as a basis for future work.

Repeated Measurements

Performing multiple measurements and outputting an extended summary of the results, such as a trimmed mean and standard deviation, could be a statistically sound way of counteracting the instability. This approach comes with a higher computational cost, which would incur both a larger latency (and worse user experience) and a smaller throughput of the whole system. However, there are several measures we could take to alleviate this:

- Provide preliminary results after the first measurement, so that students receive feedback quickly, even if it might be subject to change later.

- Abort repeated evaluations if the submission fails for reasons different than an exceeded time limit (typically a wrong output) in at least one of the tests.
- If a test exceeds the time limit by a large margin, abort repeated measurements.

These mitigation steps, along with being able to use more cores than if we relied on a single measurement, might lead to a better overall throughput of the system and a tolerable increase in latency.

Instruction Counting

The count of executed instructions is a stable measure of execution time that we could use for judging instead of the CPU time or wall clock time. However, there are multiple considerations:

- The execution time of different types of instructions can vary. A possible implementation could assign a cost to each instruction type and then calculate the sum of the instruction counts multiplied by their respective costs.
- To our best knowledge, there is no practical way of measuring the exact number of executed instructions divided by type. A sample could be obtained by profiling the submission using the `perf` tool. If we chose to sample too frequently, we might unnecessarily prolong the evaluation.
- Sampling parallelized programs might impact their performance.
- The execution costs of instructions, and even the instruction set itself, can vary with different processor models.

Using Single-board Computers

Using a large number of single board computers (such as the Raspberry Pi), where each evaluates a single submission might prove to be a viable alternative to using a server machine. We could achieve a large degree of parallelism without any concern about measurements influencing each other. The total power consumption of these computers should be comparable to that of a single server computer. However, the fact that most single computers use a different instruction set than x86 might restrict the set of usable exercises.

3. Selection of a Scheduling Algorithm

In this chapter, we evaluate various scheduling (load balancing) algorithms in the context of automated programming assignment evaluation.

3.1 Problem Categorization

Scheduling is the problem of assigning work units (jobs) to execution units (worker machines in our case). Every job has a processing set – a set of machines that are capable of processing it. There are variants of the problem with different requirements on the worker machines, processing set restrictions and other characteristics.

In this section, we list the subcategories of the scheduling problem that were studied in previous work. The categories are based on whether the problem is online or offline, on the characteristics of the worker machines and the processing sets of the jobs, the characteristics of deadlines, the existence of explicit priorities and on the ability of the scheduler to interrupt jobs and estimate the processing times. Then, we attempt to assign our use case to these categories. We shall use this knowledge to select the algorithms to be evaluated in subsequent experiments.

3.1.1 Online/Offline Scheduling

In online scheduling, the algorithm does not have access to the whole input instance as it makes decisions[25], as opposed to offline scheduling where the input is available immediately. This model is appropriate for our use case because we need to react to assignment submissions immediately.

Two variants of online scheduling are studied – over list and over time. In scheduling over list, jobs are presented to the algorithm in a sequence. The scheduling decision has to be made immediately, it is irrevocable and after it is made, another job is presented to the algorithm.

In scheduling over time, jobs arrive on their release date and can be scheduled at any time after their arrival. Typically, actions are taken on decision points – points in time when a job arrives or when a job is finished. The problem of scheduling assignment solutions to worker machines corresponds to the time-based variant – jobs that enter the system can be held in a queue until a scheduling decision is made and other jobs might arrive in the meantime. However, an algorithm for list-based scheduling could also be used in an evaluation system without any problems.

3.1.2 Preemption

The ability to interrupt long-running jobs is very useful in scheduling algorithms. When it is not available, situations where a machine needed by a specialized job is blocked by a particularly long job that could have been processed somewhere else can easily arise.

At this moment, preemption is not supported by ReCodEx. The difficulty of implementing it depends largely on the exact requirements on the guarantees provided by the interruption operation.

A simple cancellation of the current job is a rather simple feature to add, but without being able to resume it, it is certain we will lose progress, which could be a substantial setback for long-running jobs. Our scheduling algorithm would also have to make sure that a job cannot be interrupted ad infinum.

With little added effort, we could let the worker machines keep the state of interrupted jobs, which would prevent losing all of the progress. For example, if the submitted program is run against multiple test inputs, we could keep the results of measurements that are already finished. However, we could still lose a substantial amount of time by interrupting a measurement. Moreover, the job would have to be resumed on the same machine (and the scheduler would need to keep track of the fact).

Suspending measurements so that they can be resumed immediately and at the same point of their execution would be a challenging task due to its probable impact on measurement stability. A more achievable goal would be preemption by cancelling the current task (atomic part of the job) and starting that task over when the job is resumed.

Due to these difficulties, we will mainly be concerned with the non-preemptive variant, but if it shows that preemption brings interesting benefits, it will also be considered.

3.1.3 Clairvoyance

In the clairvoyant variant of the scheduling problem, the algorithm knows the exact processing time of each job on arrival[26]. Naturally, this makes it possible to schedule jobs more efficiently. In the non-clairvoyant variant, the algorithm knows nothing about the processing times. A similar subcategory of the problem exists, where the algorithm has an estimate of the processing time. This is called semi-clairvoyant scheduling.

Typically, the longest part of job processing in ReCodEx is compilation and execution of code submitted by students. The time required for this is highly unpredictable – there are submissions that fail to compile and therefore are evaluated very quickly, and there are those that run until they deplete the time limit. This is especially true for exercises in languages that allow pre-computing some values during compilation or feature complex compile time metaprogramming, such as C++.

Despite this fact, we might be able to estimate the processing time well enough e.g., by analysing run times of previous similar jobs. Therefore, we should definitely evaluate algorithms for the semi-clairvoyant variant, along with those for the non-clairvoyant variant.

3.1.4 Processing Set Characteristics

Specialized algorithms exist that solve the online scheduling problem for jobs whose processing sets satisfy additional criteria[27].

The **Inclusive** variant requires the processing sets of any two jobs to be comparable (i.e., one must be a subset of the other). This is a criterion we cannot satisfy in case there are multiple specialized workers, such as one meant for GPU computations and another with a NUMA setup for parallel CPU computations. With a worker pool like this, jobs meant for either of the two specialized machines could not be processed on the other machine – in other words, we would receive two disjoint processing sets.

In the **Nested** variant, the processing sets of every pair of jobs must be either disjoint (no common elements) or comparable. Such conditions could be achieved with special care by the administrator of the system. Without that, it could easily happen that there is for example a set of machines capable of evaluating highly parallelized submissions, another set of machines that can evaluate submissions in Java, and these sets have a non-trivial intersection. In this case, a pair of jobs where one requires a worker that can run parallel programs and the other needs a Java environment violates the criterion.

However, the requirements of the nested variant can be easily satisfied by a setup where there is a large pool of general purpose workers and a handful of specialized worker groups that do not accept any of the regular jobs. The requirements hold even when the general purpose group is composed of workers with different hardware groups, provided that jobs that allow multiple hardware groups are not issued.

A set of machines satisfies the **Tree-hierarchical** criterion if it can be arranged into a tree so that the processing set of each job is a path from the root to some node. It follows that the machine in the root must be able to process any job. Sadly, we cannot guarantee these conditions.

A special case of the Tree-hierarchical variant exists that we mention for the sake of completeness. In this setup, the machines form a single chain and the processing set of every job is a segment of this chain that starts with the first node. This is called a **Grade of Service** processing set structure.

Interval processing sets require that the machines can be linearly ordered so that the processing set of any job is a continuous interval in this ordering. A setup with a general-purpose group of workers and multiple specialized groups where each job can be accepted exactly by the workers from one particular group satisfies this criterion. However, it would be difficult to determine whether or not the criterion holds under more complicated eligibility constraints (for example, if we wanted to filter the processing sets by additional criteria such as the allowed number of parallel threads).

Since the current job routing model used in ReCodEx requires an algorithm that supports arbitrary processing sets, we will mainly be looking for those. It is also possible to restrict the job routing model (and the diversity of the worker pool) so that it conforms to the nested or interval variants of the problem. If we find any interesting results for these varieties (such as a hypothetical algorithm that would perform well on interval processing sets), we will consider this trade-off.

3.1.5 Machine Characteristics

In a setup with **related** machines, each machine has a speed and the processing time of a job on a machine can be obtained by dividing the length of a job with the speed of the machine. A setup has **identical** machines if the speeds are equal for all the machines. In the **unrelated** case, the times can vary unpredictably on each machine[28]. For our problem, we should mainly be concerned with algorithms for the unrelated case, because our worker pool can contain machines of different processing power. However, restricting the processing sets of jobs to machines with the same speed is also a viable option. To achieve this, assigning each job to exactly one hardware group would suffice.

3.1.6 Job Deadlines

Although there are no inherent deadlines in the context of a programming assignment evaluation system, we could determine them e.g., using an estimated processing time of the jobs. This could help the subjective responsiveness of the system – the scheduler could prioritize short jobs while allowing a longer waiting time for long jobs.

3.1.7 Explicit Priorities

In numerous scheduling problems, explicit priorities can be given to jobs when they are entering the system. These priorities are then used to determine the order in which the jobs are scheduled. An example of such priority is the niceness factor given to UNIX processes by users.

In our case, no such priority setting exists, even though there are requirements such as that shorter jobs should be processed as soon as possible and jobs that are expected to take minutes can be postponed. In a commercial assignment evaluation system, we might need to establish multiple grades of service based on billing rates. With such setup, lower grades of service would get a lower priority. Such use cases are however out of the scope of our research.

3.2 Analysis

In this section, we describe the requirements for a scheduling algorithm and propose a method of comparing these algorithms experimentally.

3.2.1 Requirements

Principally, there are two main performance metrics for scheduling algorithms: latency and throughput. Latency is important for interactive workloads and throughput is valued in batch workloads.

We consider giving feedback to students quickly the main benefit of a system for automated evaluation of programming assignments. Thus, optimizing latency should be preferred over optimizing throughput (but we should be vary of algorithms that grant a small improvement of latency at the cost of a large negative effect on throughput).

However, the acceptable latency varies with the processing time of the jobs. It is possible to delay jobs that take several minutes by a whole minute without a negative effect on the user experience, but the same cannot be said for jobs that take just a few seconds.

The time a user is willing to wait for an evaluation to finish is rather difficult to estimate. For a regular web page, the tolerable response time is about 5-10 seconds[29]. In our case, the motivation to see the results is higher than for a web page, which implies that the user might be willing to wait longer. Also, it is reported that providing the user with a progress feedback prolongs the tolerable waiting time (between 15-46 seconds).

In ReCodEx, the user receives information about the progress after the evaluation starts, but not while the solution is waiting in a queue. We could however provide the users with the number of solutions in the queue and an estimate of the waiting time for a better user experience.

Despite this, if an evaluation takes more than a minute to finish (including the queue time), it is unlikely that a user will wait for it – it is more likely they will switch to a different task and return to see the results later. Therefore, we should strive to keep the queue time in jobs that take tens of seconds short, even if it means postponing longer jobs by minutes.

3.2.2 Objective Function

Many different flavors of the online scheduling problem are studied. In this section, we explore currently researched objective functions to select those that align well with our requirements. Surveying the objective functions is also important for researching prior art, since literature typically focuses on the optimization of a single metric.

One of the most researched objectives in scheduling is the **makespan** – the total time it takes to process a whole workload. This has an obvious correlation with the total throughput of the system, but individual jobs can be delayed for a long time.

The **flow time** is the time spent in the system for a job. Minimizing the sum of these times (or a weighted sum) might lead to more constrained delays for individual jobs, which would mean better response times for our system.

Tardiness of a job is defined as the difference between the completion time and the deadline. The usability of this metric depends on our choice of deadlines. Tardiness seems similar to latency, but it is often used in literature about scheduling. When discussing this particular objective function, we prefer using tardiness instead of latency throughout this text. Typically, an aggregate of tardiness over all jobs (such as the arithmetic mean) is used to evaluate scheduling algorithms.

Stretch of a job is defined as the ratio between the time spent in the queue and the processing time. A benefit of this metric is that it accounts for the length of the job itself and therefore allows longer waiting times for long jobs. Exactly like in the case of tardiness, an aggregate of the stretches of all jobs is used to compare scheduling algorithms.

It would also be possible to define a custom metric based on a function of the processing time and queue time that captures the aforementioned requirements on the wait time for users based on the processing time of the job. However, this

can also be achieved by setting appropriate deadlines and using the tardiness of jobs as a metric.

From our list of metrics, we see that the flow time, tardiness and stretch are all closely related to the latency, as opposed to the makespan, which corresponds to the throughput.

When researching the problem in literature, we should be primarily concerned with the first group (focused on latency). In experimental evaluation, we should observe both the makespan and some subset of the metrics related to latency.

3.2.3 Experiment Methodology

To select the right scheduling algorithm, we will perform an experimental evaluation. In this section, we attempt to find the optimal methodology for such an experiment.

We shall observe the selected metrics for each algorithm so that we can compare their performance. Each algorithm will be evaluated on a set of workloads – sequences of jobs with timestamps that specify when they should be presented to the system. The experiment will also be repeated on multiple sets of workers with varying sizes to get a better understanding of how the algorithms handle larger worker pools.

Since this text is mainly motivated by the ReCodEx system, we shall use parts of it in our experiment. The main benefit of this is that it already has a clearly defined API for queue management. We can use this to lessen the amount of programming needed to create a testing environment. Moreover, using the queue management API in ReCodEx lets us incorporate the best scheduling algorithms into the system after we finish our evaluation.

For an easily reproducible experiment, we should use some degree of simulation to avoid setting up the entirety of the ReCodEx system, along with writing a script that emulates job submissions at scheduled times. Measuring with the whole system up would introduce multiple problems, for example that the results would contain noise unrelated to the efficiency of the load balancing algorithm (such as delays caused by network communication), and that the measurements would take much longer than when we use simulation.

Our experiment will be performed by a script that directly uses the scheduling code of the ReCodEx broker and simulates incoming jobs based on a structured description, using a selected queue manager implementation. The script will also support configuring the set of worker machines.

3.3 Related Work

In this section, we research online scheduling algorithms with respect to the categories listed in Section 3.1.

In literature, online algorithms (those that only know a part of the input with each decision) are evaluated using a measure called the competitiveness ratio. For scheduling, the competitiveness ratio is defined as the ratio between the worst-case length of the schedule produced by the algorithm being evaluated and the worst-case length of the schedule produced by an optimal algorithm. This number is often relative to other factors such as the number of worker machines.

Since we are going to evaluate the algorithms experimentally, competitiveness ratios are not particularly interesting for us and we will not focus on them in our survey.

3.3.1 Time and List Based Scheduling

It seems that the time-based variant of the online scheduling problem is not very well researched, despite its practical applications. A heuristic approach for minimizing the makespan called OAGM (Online Algorithm based on Greedy algorithm and Machine preference) has been described[30] that uses a single queue of incoming jobs ordered by three criteria (the order is different for each worker):

- smallest label first (the label of a job on a particular worker is its position in a sequence of queued jobs, ordered by their processing times for the worker, smallest to largest),
- least flexibility job first (a job is less flexible than another if its processing set is a proper subset of the processing set of the other job) and
- longest processing time first.

Note that in the case of identical machines (a job takes the same time, regardless of the worker processing it), sorting by label degrades into sorting by shortest processing time. The other two rules will then only be used to break ties in that case.

If a job is due to be sent to an idle worker and there are multiple candidates, the job is sent to the worker with the smallest sum of processing times of queued jobs that can be processed by that worker. This number can be interpreted as a potential load factor of the worker.

This algorithm is expanded upon by a meta-heuristic algorithm (called meta-OAGM) that maintains a job queue for each worker and tries to iteratively improve the schedule by randomly moving jobs from the most loaded worker to another with a probability based on the load of each worker. This approach is based on simulated annealing, a probabilistic technique for approximating the global optimum of a function.

Most remaining literature is concerned with the list-based variant of the problem. We will survey the results in the following sections.

3.3.2 Preemption

The main benefit of being able to interrupt running jobs is that we can pause a job in order to allow a shorter job to complete quickly. A basic algorithm for scheduling with preemption is SRPT – Shortest Remaining Processing Time[31]. The idea of this algorithm is that incoming jobs are placed into a queue ordered by expected time to completion. If a queued job has a lower expected time to completion than some job that is already being processed, the active job is interrupted, placed into the queue (with a decreased time to completion, because the worker has been processing it for some time), and the queued job is started instead. It is evident that this helps achieve a better flow time (and thus improves the latency of the system).

A drawback of the SRPT algorithm is that it does not take the cost of preemption into consideration. In ReCodEx, interrupting a job would require cancelling the current task, which means losing a certain amount of work – even tens of seconds in some cases (e.g., costly compilation or measurement tasks). Additionally, it is difficult to predict this cost. A way to counteract this cost could be allowing some time before resuming an interrupted job so that additional short jobs can be scheduled in the meantime.

Schedulers in operating systems typically switch processes frequently to give each a fair share of the computing power (the time each process is allotted is called the quantum). This approach would be problematic with our implementation of job interruption – long tasks could keep being interrupted repeatedly, causing an unnecessary delay of the job. However, we could still take inspiration from algorithms such as multi-level feedback queue scheduling, where processes that take long to process are gradually moved to queues with a lower priority and a longer quantum.

3.3.3 Clairvoyance

It is evident that not knowing how long a job will take to process makes it much more difficult to schedule it efficiently, since we cannot calculate the exact load of a machine. Non-clairvoyance also disqualifies algorithms such as SRPT, that need to know the processing time. A similar algorithm, SETF – Shortest Elapsed Time First exists for the preemptive case[32], that does not need to know the processing times. Naturally, this is also the case for scheduling algorithms used in operating systems, such as multi-level feedback queues.

For the semi-clairvoyant variant of the problem, where we only have approximate knowledge of the processing times, it has been shown that SRPT works reasonably well and a new algorithm similar to multi-level queues has been studied[33].

3.3.4 Processing Set Characteristics

In the case where the processing sets are arbitrary, it has been shown that an algorithm that assigns jobs to the least loaded eligible machine is close to being optimal[34] with respect to the total makespan.

An implicit requirement of this is that we must be able to estimate the job processing times in order to determine which machine is the least loaded one.

For the case with nested processing sets, a makespan-minimizing algorithm[35] exists that requires the jobs to have equal processing times. The basic idea of the algorithm is scheduling the jobs with the least flexibility first. The flexibility of a job is defined as the number of workers in its processing set. This algorithm can be adapted to arbitrary processing sets as well, but we will not have any mathematically proven guarantees about its competitiveness (which does not hinder our intention to perform experimental evaluation).

For the case with interval processing sets, we have only found results for very specific instances of the problem (such as scheduling on exactly two machines) that cannot be used in our situation.

3.3.5 Machine Characteristics

Most literature is concerned with the case where machines are identical, i.e., when a job takes the same time on any machine. In the case with related machines, an algorithm that assigns jobs to the slowest eligible machine first is described[28].

3.3.6 Job Deadlines

Earliest Deadline First is a well researched approach to scheduling with deadlines. It has been used extensively in the development of real time systems. Numerous other techniques have been found in this field, but they do not apply to our use case.

3.3.7 Additional Approaches

The "Power of two choices" approach[36] is a load balancing policy featured in the Nginx web server and reverse proxy. When a job is being assigned, two workers (upstream web servers) are selected at random and the job gets assigned to the one that is better according to some metric, for example the one with a lesser load factor.

3.4 Processing Time Estimation

Some scheduling algorithms (those for the clairvoyant version of the problem and those that need an estimate of the length of the queue for the workers) require to know the processing time of a job to function. Even though we do not know this precisely, we can make an estimate based on previous similar jobs that have already completed. Of course, the margin of error will be rather high, due to the characteristics of the problem outlined in Section 3.1.

3.4.1 Estimation Formula

To accurately predict the processing times of incoming evaluation jobs, we need to categorize information about recently finished jobs efficiently. With each job, we receive the exercise identifier, the required runtime environment and hardware group and an identifier of the author. The ReCodEx core can also supply us with an upper limit on the runtime of the exercise based on the time limits of individual tests. While this information is far from precise, it can help us with estimating processing times of exercise types we have never seen before.

Based on this information, we propose the following estimation algorithm:

- Maintain three dictionary structures. The first one is indexed by runtime environment, the second one by runtime environment and exercise identifier and the third one by runtime environment, exercise identifier and author identifier. The values in these dictionaries are circular buffers of size 20.
- Whenever an evaluation is finished, save the processing time into all three dictionaries under corresponding indexes (the value is inserted into the circular buffers).
- When a new job arrives, estimate its processing time as follows:

- If there are any historical entries in the third dictionary (by author, exercise and runtime environment), return their median.
- If there are at least two historical entries in the second dictionary (by exercise and runtime environment), return their median.
- If there are at least two historical entries in the first dictionary (by runtime environment) that are smaller than the exercise limit supplied by the ReCodEx core, return their median.
- Otherwise, return the exercise limit divided by two.

We have chosen median over arithmetic mean and other possible central tendencies because it returns actual values encountered by the system, which is favorable when there are two groups of similar measurements that are very far from each other, for example. If such cases were not present in our data, using the arithmetic mean might have been preferable for queue managers that estimate the length of the queue for a worker and not of an individual job.

The thresholds used when determining whether a particular buffer of measurements should be selected were found by evaluating the algorithm on a testing data set.

The limit on the size of the circular buffers is necessary to keep memory usage from continuously growing as long as the system is online, and also to prevent old processing times from influencing predictions too far into the future. The exact value of the size limit (20) was chosen empirically, based on preliminary test runs of the estimation program – lower values caused a decline in accuracy and the improvement of higher values was negligible.

We conjectured that using a smaller buffer size for the last level dictionary (by author, exercise and runtime environment) might improve prediction accuracy, because subsequent submission are typically either very similar or faster, but we failed to verify this conjecture. Therefore, we did not use this optimization to keep the algorithm simpler.

3.4.2 Evaluation of the Estimation Formula

To evaluate the quality of the estimates, we let the algorithm process historical data and then compared the predictions with the actual results. We then analyzed the relative errors of the predictions, which were defined as $100 \times (T_{prediction} - T_{actual})/T_{actual}$ – a negative error means that the estimate was too low and vice versa. A quick analysis of the errors for the whole dataset revealed that negative errors are more frequent than positive ones (74898 vs. 61260) and that the relative error was smaller than 10% for 57% of the observations and smaller than 20% for 82% of the observations.

After an inspection of histograms of the relative errors grouped by actual processing times (Figure 3.1), we see that the number of overestimations by more than 100% is not negligible (around 7% of the total number of observations). Also, for very short jobs (less than 100 milliseconds), we overestimate the processing time very often. This is not surprising, since short processing times often indicate a failure, which is difficult to predict based on the available information. This phenomenon is also likely to have caused the notably large frequency of cca. 100% negative errors (the estimator returned a very small prediction) in jobs longer than 50 seconds.

In summary, the estimation algorithm seems to perform rather well, despite infrequently overestimating the processing time by a margin of several hundred percent. It is certainly suitable for usage in our load balancing algorithm evaluation experiment. In the future, more sophisticated algorithms could be devised, for example using machine learning techniques.

3.4.3 Estimation in Simulated Experiments

As mentioned in Section 3.2.3, we will evaluate queue management algorithms in a simulated environment with simplified job data that does not necessarily have to originate from actual records of ReCodEx traffic.

Problems are very likely to arise if we test a queue manager that uses our estimation strategy on simple, more predictable jobs – its accuracy will be different than it would be on real world data.

As a countermeasure, we decided to implement an imprecise estimator for the simulation that adds a random error to the actual processing time of the job. The procedure for the error generation is as follows:

- Determine whether the error will be positive or negative using random sampling based on the values from our dataset.
- Generate a random integer `i` between 0 and 99.
- Take a table of selected percentiles of the prediction errors in our dataset (Table 3.1) and select a pair of percentiles between which `i` fits (there are separate rows for positive and negative errors). The way i is selected guarantees that the probability that a span between two percentiles is chosen corresponds to its size.
- Generate a random number from a uniform distribution between the values of the two percentiles.

It is evident that the errors obtained using this procedure will have a distribution that roughly approximates that of the errors encountered in our dataset. A downside of this approach is that the errors are fixed and do not change over time as the estimator receives more information.

Table 3.1: Selected percentiles of empirical estimation errors used to determine estimation errors in synthetic workloads for load balancing algorithm evaluation

	5th	10th	20th	40th	60th	80th	95th	100th
Positive Error	0.2%	0.5%	1.1%	3.1%	11.3%	68.6%	963.2%	63529.0%
Negative Error	0.3%	0.6%	1.4%	4.0%	12.0%	37.1%	83.8%	100.0%

3.5 Custom Algorithms

In this section, we explore the possibility of modifying two well-known online load balancing algorithms that could not be directly applied to our use case.

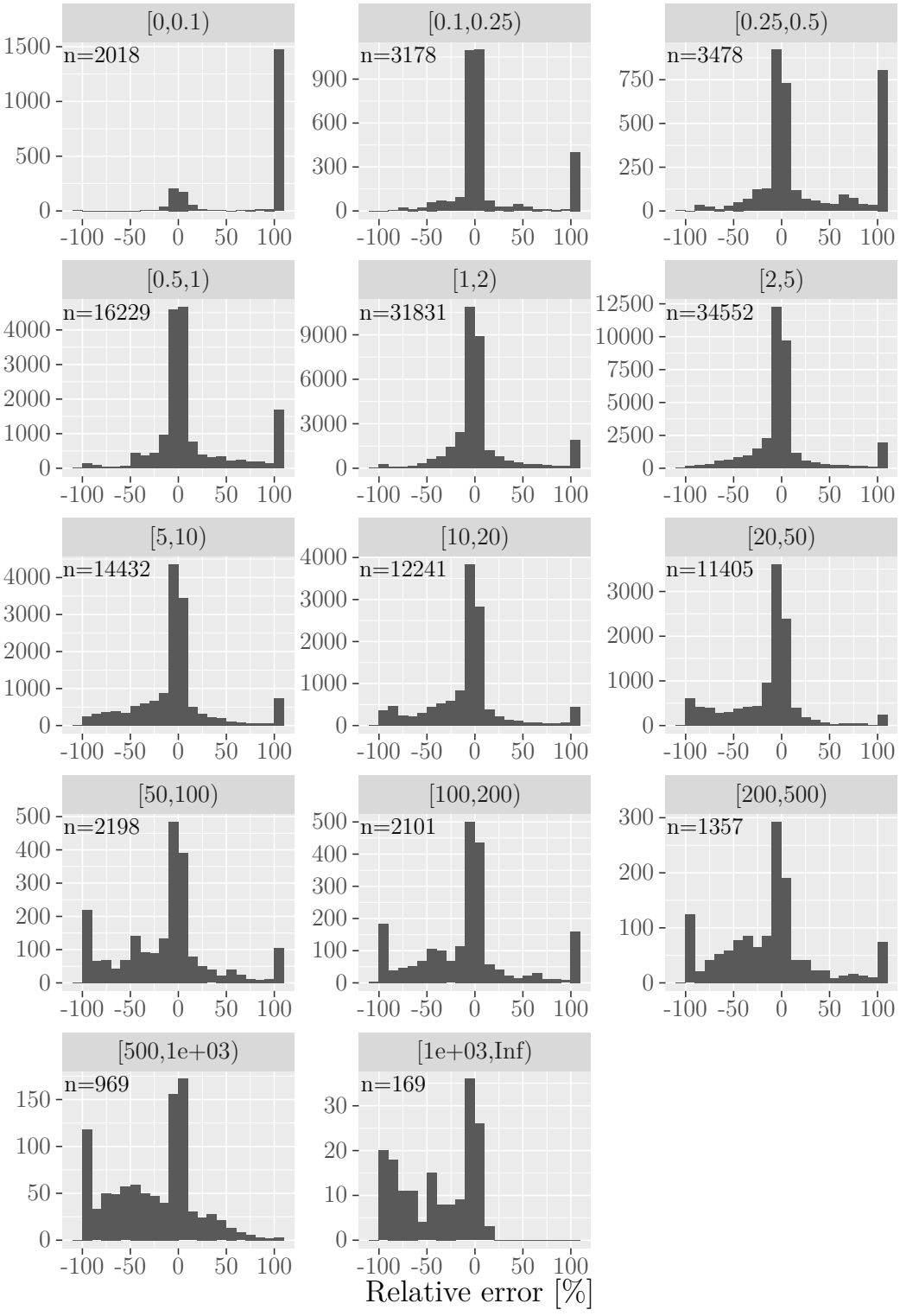

Figure 3.1: Histograms of relative estimation errors divided into facets by job processing times (in seconds). The rightmost bin of each histogram contains all errors larger than 100%.

3.5.1 Earliest Deadline First Approach

Scheduling jobs in an order of increasing deadlines is frequently used in development of real time systems and other applications. In our situation, no deadlines are given explicitly. However, we can set a deadline based on the expected processing time of the job (an estimate based on recorded processing times of similar jobs) and allow additional time for longer jobs (based on the requirements on fast feedback described in Section 3.2.1).

We propose the formula to obtain a deadline based on the time of arrival and estimated processing time as follows:

- Time of arrival + the estimated processing time for jobs shorter than 15 seconds. The threshold is based on the fact that users are likely to wait for actions that take 5-10 seconds[29] and that assignment authors are more motivated to see the results than common visitors of a web page.
- Time of arrival + the estimated processing time + 15 seconds for jobs shorter than 45 seconds. A regular web page visitor is unlikely to wait until an action is completed longer than 45 seconds[29], but assignment authors might be willing to wait longer. We expect that the added 15 seconds will help the scheduler prioritize very short jobs.
- Time of arrival + twice the estimated processing time for jobs longer than 45 seconds. Since users are likely to switch to another task while waiting for the submission to be evaluated, we can allow a rather large portion of time for it to be spent in a queue while shorter evaluations are performed.

Deadlines set like this should ensure that short jobs are processed as soon as possible, but longer jobs do not wait in the queue indefinitely.

3.5.2 Multi-level Feedback Queue Approach

In its standard form, the multi-level feedback queue algorithm maintains multiple queues, each of which has a numeric level. Every incoming job is placed into the top-level queue. At some point, it is dequeued from the queue and a worker starts processing it. If it is not completed within a certain time period (the quantum), it is preempted and placed into a lower-level queue, which has a longer quantum. The jobs in the lower levels only get processed when the upper queues are empty.

A similar approach[33] was presented for semi-clairvoyant scheduling with preemption on a single machine. Each job is assigned a level – an integer l such that the estimated processing time of the job is between 2^l and 2^{l+1}. At any given time, jobs from the lowest level are processed, with one exception: if there is a single partial job (a job that has been processed for a time but was preempted) in the second lowest level and no total jobs (jobs that have not yet been processed), it is processed first. While it is not straightforward to extend this approach to a setup with multiple machines, we can use the idea of leveraging semi-clairvoyance to improve the multi-level feedback queue algorithm. As a side note, if we took away preemption, the algorithm would degrade to a simple shortest job first policy.

Standard multi-level feedback queue scheduling requires preemption to work. We propose a modification where jobs are assigned queue levels based on their

estimated processing time (similarly to the aforementioned semi-clairvoyant algorithm). The resulting algorithm becomes similar to an older approach, multi-level queue scheduling, since we miss the feedback that a job failed to complete within the given time quantum. An obvious drawback of our modification is that it does not prevent starvation (a job being held in a queue indefinitely). In fact, the result is a slightly more convoluted shortest job first policy.

Our way to remedy this is by adjusting the way queues on different levels are processed. Instead of waiting for all the queues on higher levels to empty, we determine a share of processing time for each queue level. The queue manager keeps a history of recently dispatched jobs and selects queues for assignment in a way that the proportions of usage levels determined from the active queue levels are maintained.

This modification could prevent starvation for long-running jobs while maintaining a low response time for most short jobs. However, there are multiple parameters that need to be fine tuned: the exact boundaries for assigning jobs to queue levels, the weight of each level to be used to determine its share of processing time and the length of the dispatched job history to be kept.

3.6 Evaluation

Our survey of scheduling algorithms provided us with a variety of possible approaches that we will proceed to evaluate. We decided to exclude algorithms that require preemption (such as SRPT and SETF), because implementing it in ReCodEx would be difficult and it may show that non-preemptive approaches work sufficiently well.

3.6.1 Algorithm Notation

We characterize the algorithms that will be evaluated using three variables:

- whether they maintain a separate queue for each worker and assign jobs immediately, or delay the assignment until a worker is available,
- the exact algorithm of worker selection (such as `fcfs` for first come, first served), and
- the method used for processing time estimation, where applicable.

To denote an algorithm to be evaluated, we use a shortened notation where these variables are divided by slashes, for example `n/rand2/queue_size`. The first part contains either `n` for multi queue algorithms and `1` for single queue algorithms. The other two parts are simply identifiers of the algorithm and processing time estimator names.

When the estimator name is omitted from the algorithm description (e.g., `n/rand2`), we are just referring to the algorithm and the processing time estimation mechanism is not important. For algorithms that do not employ estimation at all, we fill in a dash as the estimator name (e.g., `n/round_robin/-`)

3.6.2 Evaluated Algorithms

The current load balancing algorithm (`n/round_robin/-`, a simple round robin over all workers) in ReCodEx belongs to the first category (multiple queues, immediate dispatch), along with assigning incoming jobs to the least loaded worker (`n/load`) and the "Power of two choices" randomized algorithm (`n/rand2`). We will employ three ways of estimating the load of the workers:

- `queue_size` – simply counting the jobs and using the total as the load size,
- `oracle` – an estimator which has access to the simulation data and returns the exact right processing time, and
- `imprecise`, which was described in Section 3.4.3.

This gives us a total of seven algorithms to evaluate – three variants of both `n/load` and `n/rand2` (`n/load/oracle`, `n/rand2/queue_size`, etc.), and the original algorithm, `n/round_robin/-`.

The single queue (delayed dispatch) category contains two broad approaches. One is based on a single priority queue of jobs with various policies. We will evaluate the following priority policies:

Possibly the most straightforward algorithm in the delayed dispatch category is based on a single priority queue of jobs. There are many possible policies for assigning priorities to jobs. From these, we will evaluate the following:

- `1/fcfs/-` – first come, first served, also known as earliest time of arrival first,
- `1/oagm` – Online Algorithm based on Greedy algorithm and Machine preference, the policy mentioned in Section 3.3.1, which uses multiple criteria derived from the processing time and flexibility of jobs to order the queue,
- `1/spt` – shortest job first (based on previous processing times),
- `1/edf` – earliest deadline first (the modification presented in Section 3.5.1), and
- `1/least_flex/-` – least flexibility job first.

From these algorithms, three employ processing time estimation (`1/edf`, `1/spt` and `1/oagm`). These algorithms will be evaluated twice, once with the `oracle` estimator and once with the `imprecise` estimator (`1/edf/oracle`, `1/spt/imprecise`, etc.).

The multi-level queue family contains examples of more sophisticated delayed dispatch algorithms. The MLFQ algorithm depends heavily on preemption, which disqualifies it from our experiment. We could evaluate the modification described in Section 3.5.2, but it would require extensive preliminary testing to find appropriate values for its parameters. We will therefore omit the multi-level queue algorithms from our evaluation.

3.6.3 Experimental Workloads

To cover multiple use cases, we will measure the queue manager performance on various sequences of jobs with different sets of workers, which are described in this section.

The worker sets are configured manually and are not intended to change between iterations of the experiment. The job sequences, on the other hand, are randomly generated to ensure the robustness of our experiment. The job processing times are sampled from a normal distribution with a mean (μ) and standard deviation (σ) based on historical data from ReCodEx, which are depicted in Figure 3.2. Description of the runtime environments can be found in ReCodEx documentation[37] (it is worth noting that the `python` environment is used both for short programs in programming basics courses and for machine learning assignments that take several minutes to run). No rigorous tests for goodness of fit of a normal distribution on actual job processing times have been performed.

Throughout the experiment, we use the following job types:

- `common_short`, with $\mu = 500ms$ and $\sigma = 200ms$, which corresponds to the parameters of a trivial program in C, C++ or Pascal,
- `common_medium`, with $\mu = 2000ms$ and $\sigma = 500ms$, a plausible processing time for an exercise in C, C++, Python or Node.js,
- `common_long`, with $\mu = 10000ms$ and $\sigma = 4000ms$, which could be a long exercise in C++, Java or C#,
- `parallel`, with $\mu = 8000ms$ and $\sigma = 2000$, which could be a parallel workload in C++, and
- `gpu_ml`, with $\mu = 1000000ms$ (16 minutes and 40 seconds) and $\sigma = 120000$ (2 minutes), which could correspond to a long GPU-based workload, such as training a neural network.

The delays between jobs are sampled from an exponential distribution with a mean that varies with the workload types (although 100ms is a common value), which is a common assumption when modelling queuing scenarios. For some workloads, the mean delay was used to adjust the intensity of incoming jobs and thus help realize the scenario tested by the workload (e.g., fully saturating a particular group of workers).

Common and Parallel Jobs

The common and parallel workload, which is denoted as `common+para_small` and `common+para_large` in measurement scripts and results, contains jobs of the `common_medium` and `parallel` types in a 3:1 ratio (so that the makespans are similar on the common and parallel worker groups). In the small variant, there are 1000 jobs executed on 10 workers capable of processing the `common_medium` type and 1 worker capable of processing the `parallel` type.

In the large variant, there are 4000 jobs executed on four parallel workers and 40 common workers. Both variants have a mean delay of 100ms. The purpose of the workload is to show how queue managers handle job types with disjoint processing sets.

Two-phase Workload

In the two-phase workload (denoted as `two_phase_small` and `two_phase_large` in measurement scripts and results), there are 2000 jobs. The first 1000 is of the `common_short` type. The second 1000 contains jobs of both the `common_short` and `common_long` types in a 1:5 ratio (chosen empirically).

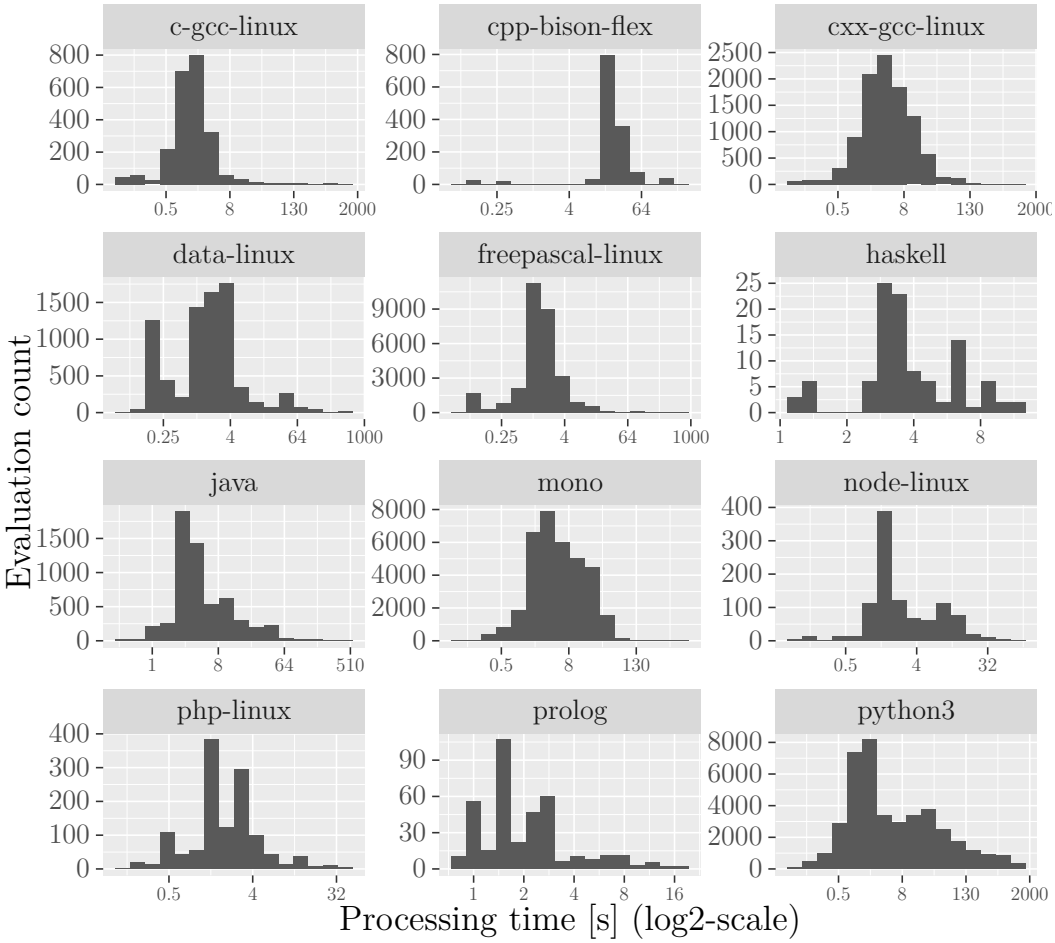

Figure 3.2: A breakdown of job processing times from ReCodEx divided by runtime environments

The workload is executed on 40 identical workers with a mean delay of 550ms in the large variant and on 4 workers with a 45ms mean delay in the small variant. The values of the mean delay were chosen empirically to make sure that the workers are not over-saturated at the end of the first phase. The purpose of this workload is to see how different queue managers react to a sudden change in the character of incoming jobs.

Long and Short Jobs

The are two workloads that employ a sequence of jobs of two different lengths. The first one is called `medium+short` and it is composed of `common_medium` and `common_short` jobs in a 1:1 ratio with a 100ms average delay. It is executed on a set of 4 identical workers. The other workload is called `long+short` and contains `common_short` and `common_long` jobs in a 4:1 ratio with a 100ms mean delay. We use 40 identical worker machines to execute this workload. The ratio of the job types is a tunable parameter that carries no particular meaning.

This kind of workloads aims to test whether the load is distributed evenly among the workers. If not, we should observe a longer makespan.

Multiple Job Types

The last workload type we will use is called `multi_type` and it contains a multitude of job types (as described by Table 3.2). There are 1000 jobs with an average delay of 100ms. The worker pool contains contains two common groups of workers, one with 6 workers and the other with 4. Apart from these two, there is a parallel group with 2 workers and a GPU group with 4 workers.

This workload is meant to test the queue managers in a setting that is closer to a real-world usage scenario than the others, with multiple kinds of jobs arriving in a short period of time. The frequencies of each type are chosen arbitrarily, without any relation to real world data.

Table 3.2: Parameters of jobs in the `multi_type` workload type

Probability	Job type
35%	common_medium (common hardware group 1)
30%	common_medium (common hardware group 2
30%	common_medium (common hardware group 1 and 2)
4%	parallel
1%	gpu_ml

3.6.4 Summary of Results

To allow evaluation of the results of the simulation at a glance, we decided to classify the jobs based on the wait time as follows:

- For jobs shorter than 5000ms:
  - "On time" if the wait time is shorter than 2000ms,

  - "Delayed" if it is shorter than 15000ms,
  - "Late" if it is shorter than 45000ms and
  - "Extremely late" otherwise

- For jobs longer than 5000ms:
  - "On time" if the relative wait time is smaller than 0.4,
  - "Delayed" if it is smaller than 3,
  - "Late" if it is smaller than 9 and
  - "Extremely late" otherwise

The breakpoints in the classification are partially chosen empirically and partially based on the wait time requirements outlined in Section 3.2.1. The justification for the split between short and long jobs is that it would be unreasonable to classify a 500 millisecond job as extremely late if it was delayed by 5 seconds (a relative wait time of 10). Analogous to this, it would not be accurate if we classified a 10 minute job as extremely late if it got delayed by one minute (which is however a substantial delay for a job that is processed in mere seconds).

After the classification, we made a plot for each workload that compares the share of each of these classes between different queue manager implementations. To reveal the development of these ratios in time, the jobs are split into 20 bins of equal size based on their time of arrival (for a workload with 1000 jobs, the first bin contains the first 50 jobs to arrive, the second bin contains jobs 51 to 100, and so on). The share of each class is then displayed separately for each bin using a stacked bar plot.

Our examination of the measurements revealed several interesting trends. First of all, most of the queue managers that use per-worker queues perform worse than others, even on simple workloads. This is demonstrated in the case of the `n/load`, `n/rand2` and `n/round_robin/-` strategies and the `long+short` workload (as shown by Figure 3.3). While other algorithms manage to process most jobs on time, the multi-queue algorithms only achieve that for a fraction of the workload. The share of "late" jobs is somewhat larger for the `n/round_robin/-` algorithm, which is much less sophisticated than the others.

A similar situation can be seen in the case of the `multi_type` workload (depicted by Figure 3.4). Here, the `n/load` algorithms perform better than the `n/round_robin/-` and `n/rand2` variants. However, the single-queue algorithms still outperform them in terms of the ratio of jobs processed on time. On the other hand, it is worth noting that most of the single-queue approaches processed some jobs extremely late, which did not happen with multi-queue approaches.

Our second observation is that the `1/spt` algorithm performs better or similarly well as the others on all measured workloads, as long as we are concerned about the number of jobs processed on time. This fact can be seen in Figure 3.5, where most queue managers initially perform rather well, but their performance drops dramatically after roughly 250 jobs, while `1/spt` still manages to process a part of the jobs on time. The `1/oagm` and `1/least_flex/-` algorithms also seem to have somewhat better results than other algorithms. In the case of `1/least_flex/-`, this should be attributed to coincidence, since this workload employs identical workloads and all jobs are considered equal by the queue manager.

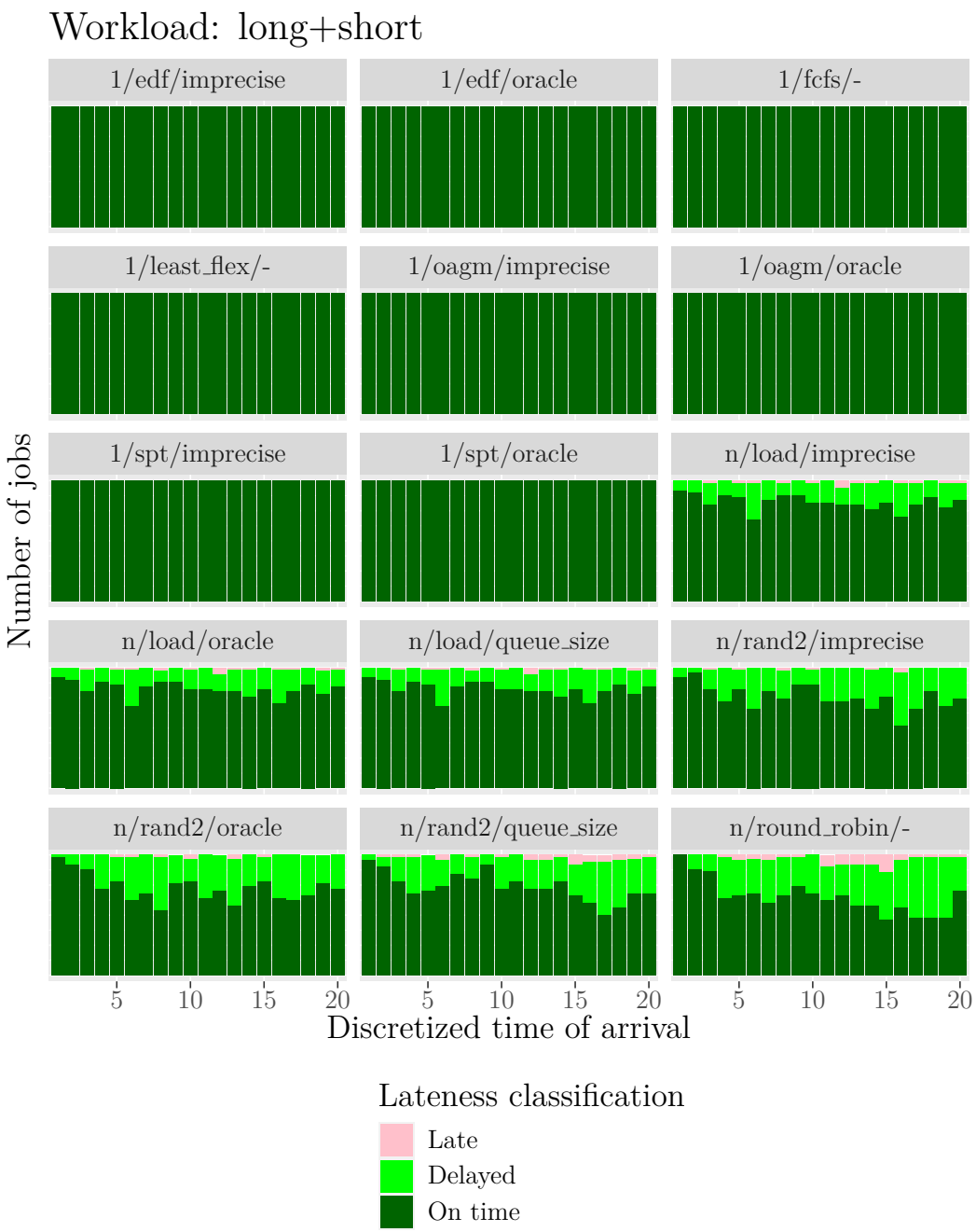

Figure 3.3: Lateness classification for each queue manager over arrival time windows for the `long+short` workload

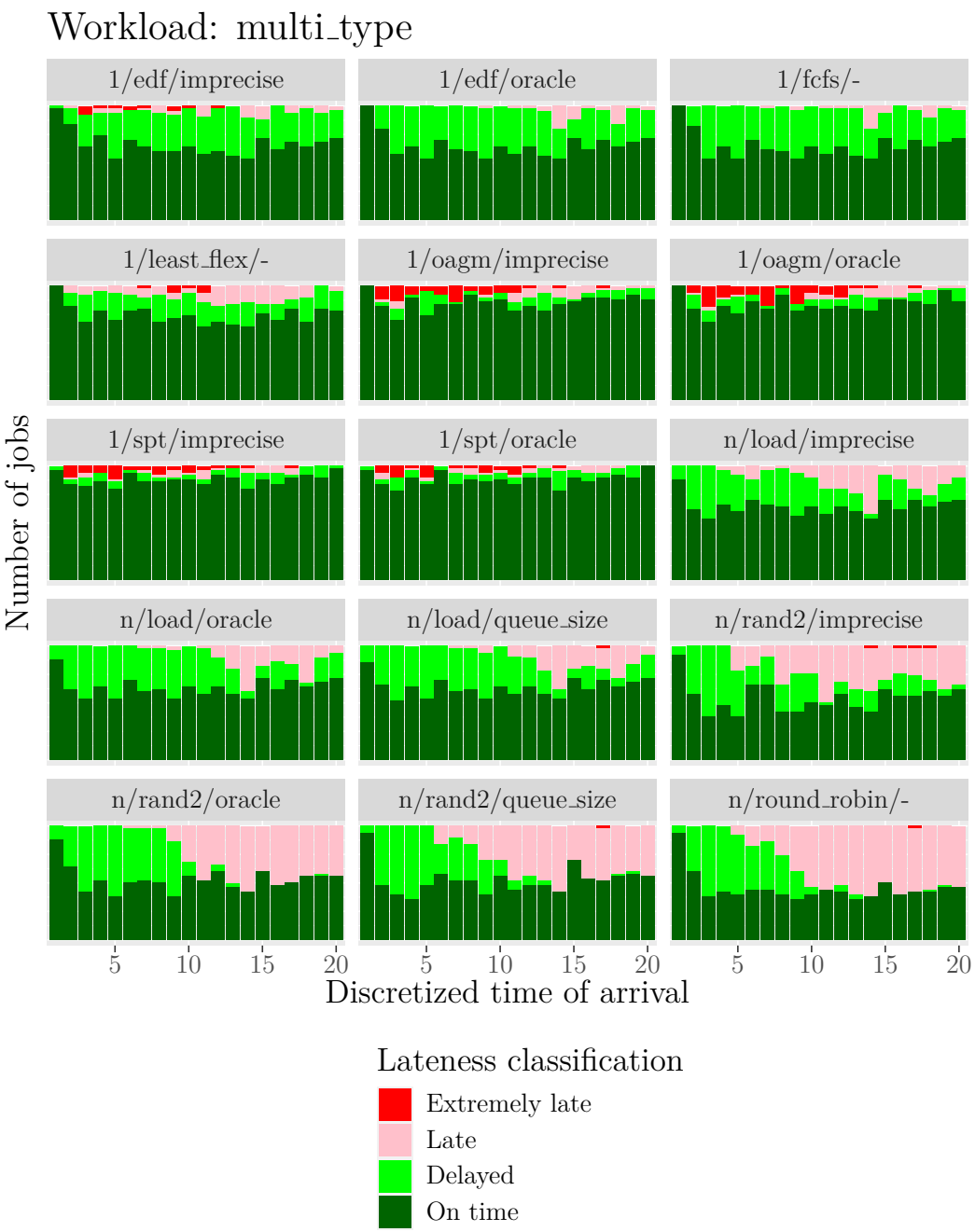

Figure 3.4: Lateness classification for each queue manager over arrival time windows for the `multi_type` workload

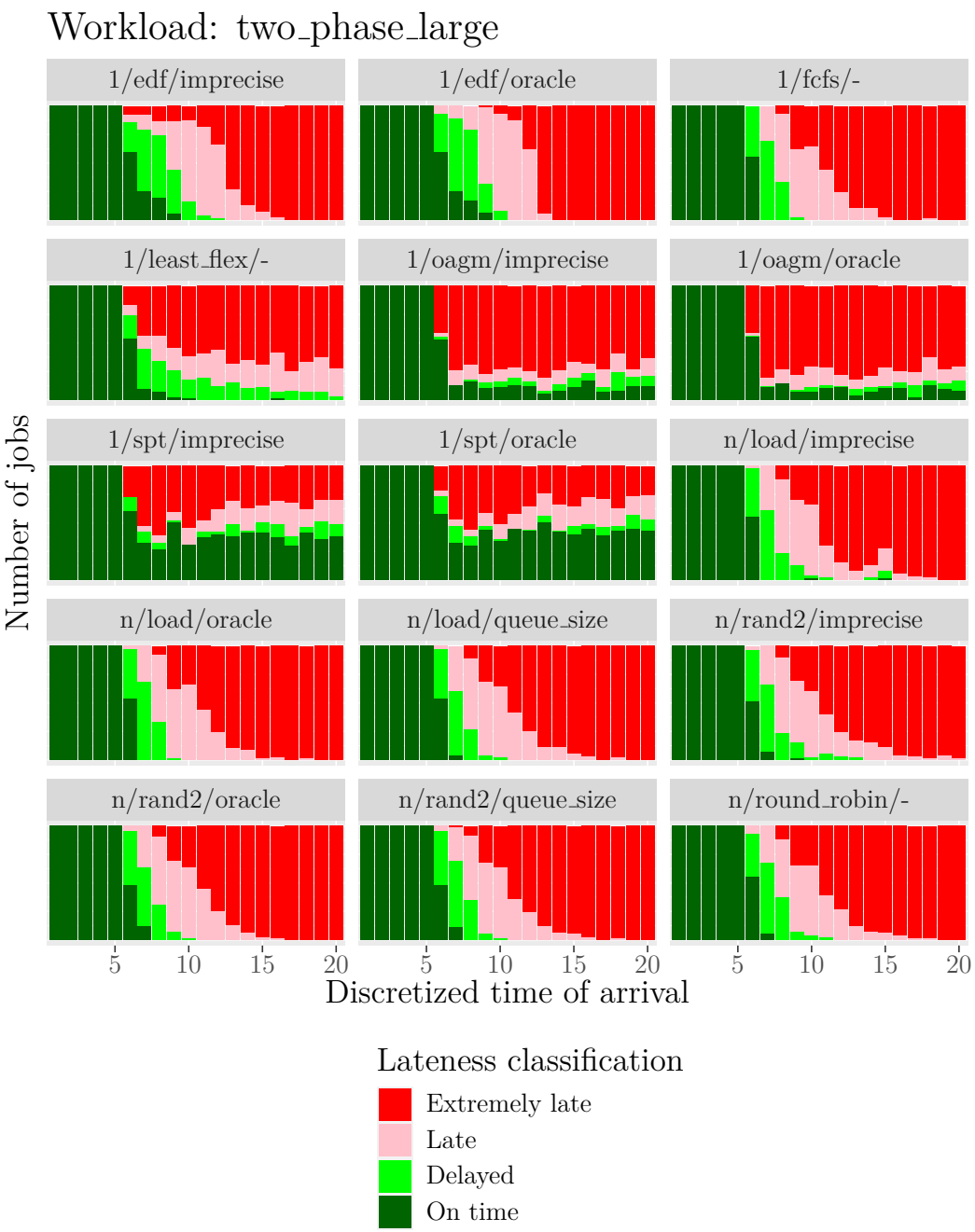

Figure 3.5: Lateness classification for each queue manager over arrival time windows for the `two_phase_large` workload

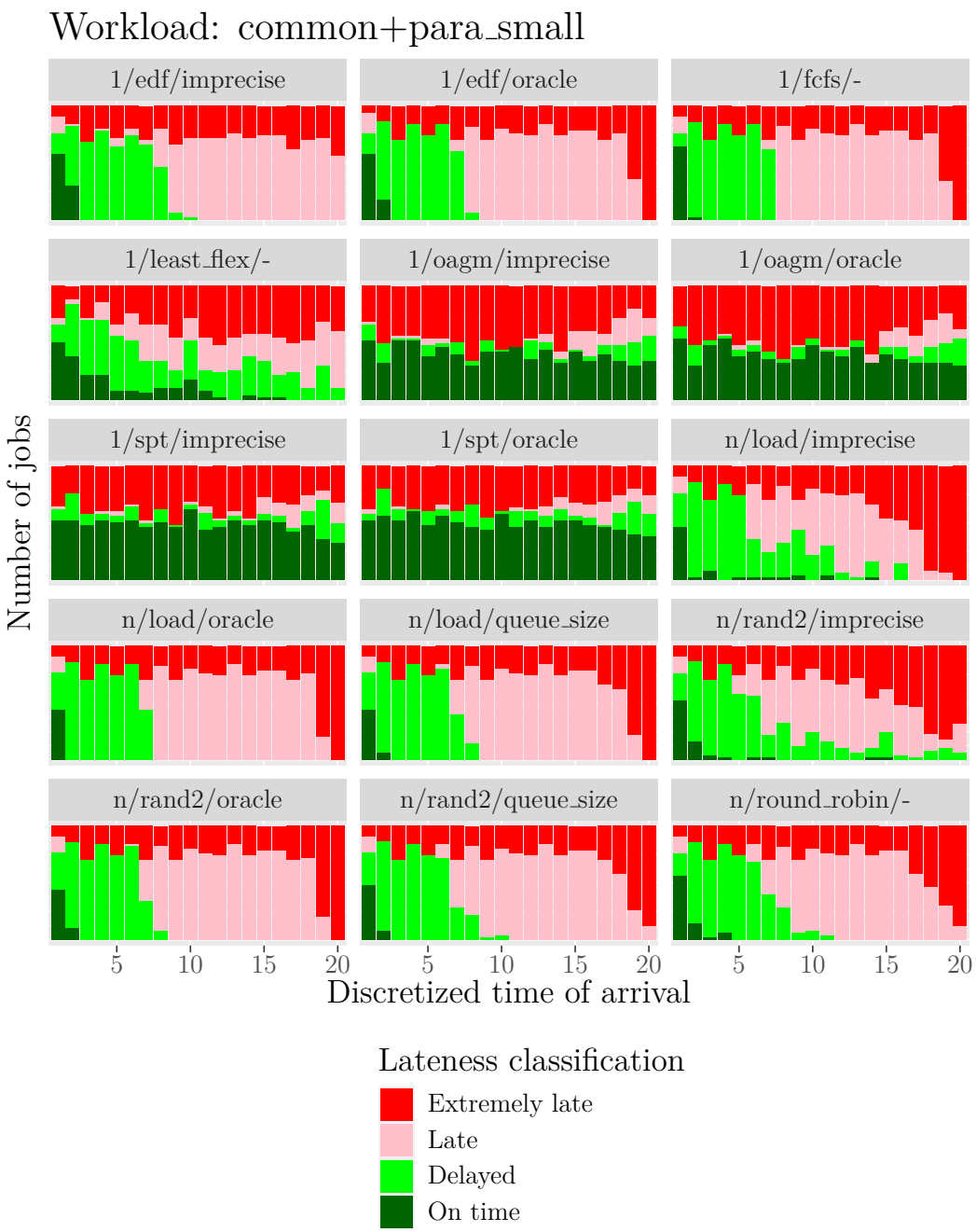

Figure 3.6: Lateness classification for each queue manager over time for the `common+para_small` workload

Another occurrence of this trend is the `common+para_small` workload, whose lateness classification can be seen in Figure 3.6. The results show `1/spt` to be the best performing algorithm, with `1/oagm` on the second place. In this case, `1/spt` prevails not only in terms of the number of jobs processed on times, but also in terms of the number of jobs that were processed late or extremely late.

The last observation we made is that the exact mechanism of processing time estimation does not cause a noticeable change in the ratio of lateness classes. However, the lateness class graphs are a rather crude visualization that is useful for a quick comparison, but it might fall short in this case.

For a more detailed insight, we plotted separate histograms of the relative waiting times for each queue manager (as depicted in Figure 3.7). In these, we have noticed that using the imprecise estimator causes an increase in cases with very large relative waiting times for `1/spt`. The other algorithms that employ processing time estimation do not seem to be affected in any notable way. In the case of `n/load` and `n/rand2`, this could be caused by the larger estimation errors being compensated for by other jobs with better estimates in each queue. It is difficult to find an explanation in the case of `1/edf` and `1/oagm`. We hypothesize that `1/oagm` is not as reliant on processing time estimation because it also uses other metrics to order the job queue. The results for `1/edf` should probably be attributed to coincidence.

The last subject of our evaluation were the makespans (total time it takes to process a workload) of the individual queue managers. A selection of comparison plots can be seen in Figure 3.8. In most workloads, the queue managers exhibit very similar makespans. An exception to this trend are the `n/load/imprecise` and `n/rand2/imprecise`, whose makespans are longer. A plausible explanation of this is that they fail to balance the load on the individual workers due to not being able to estimate the length of their queues well enough. In some cases, longer makespans can be observed for `n/rand2/oracle` as well, probably because of its inherent randomness.

It is worth noting that using the imprecise estimator does not seem to affect the makespan of queue managers that maintain a single queue nearly as much. It can be conjectured that the estimation error becomes a problem only when many estimates are summed, which is the case with multiple-queue strategies.

3.7 Conclusion

Our experiment has shown that a simple "Shortest processing time first" algorithm beats all the other approaches we evaluated. Judging by intuition, long jobs could be prone to starvation when this algorithm is used. However, our data suggests this is not the case. On the contrary, it exhibits the least number of outliers in terms of relative wait time with an ideal processing time estimator. With an imprecise time estimator, its number of outliers is still competitive.

There were cases where some other heuristic approaches showed promising results. For example, the OAGM algorithm, the "Least flexible job first" heuristic or our implementation of the "Earliest deadline first" policy. It is possible that some combination of these approaches would have interesting performance, but an exploration of such a large pool of combination is out of the scope of our research. There is also a chance that the "Earliest deadline first" approach could

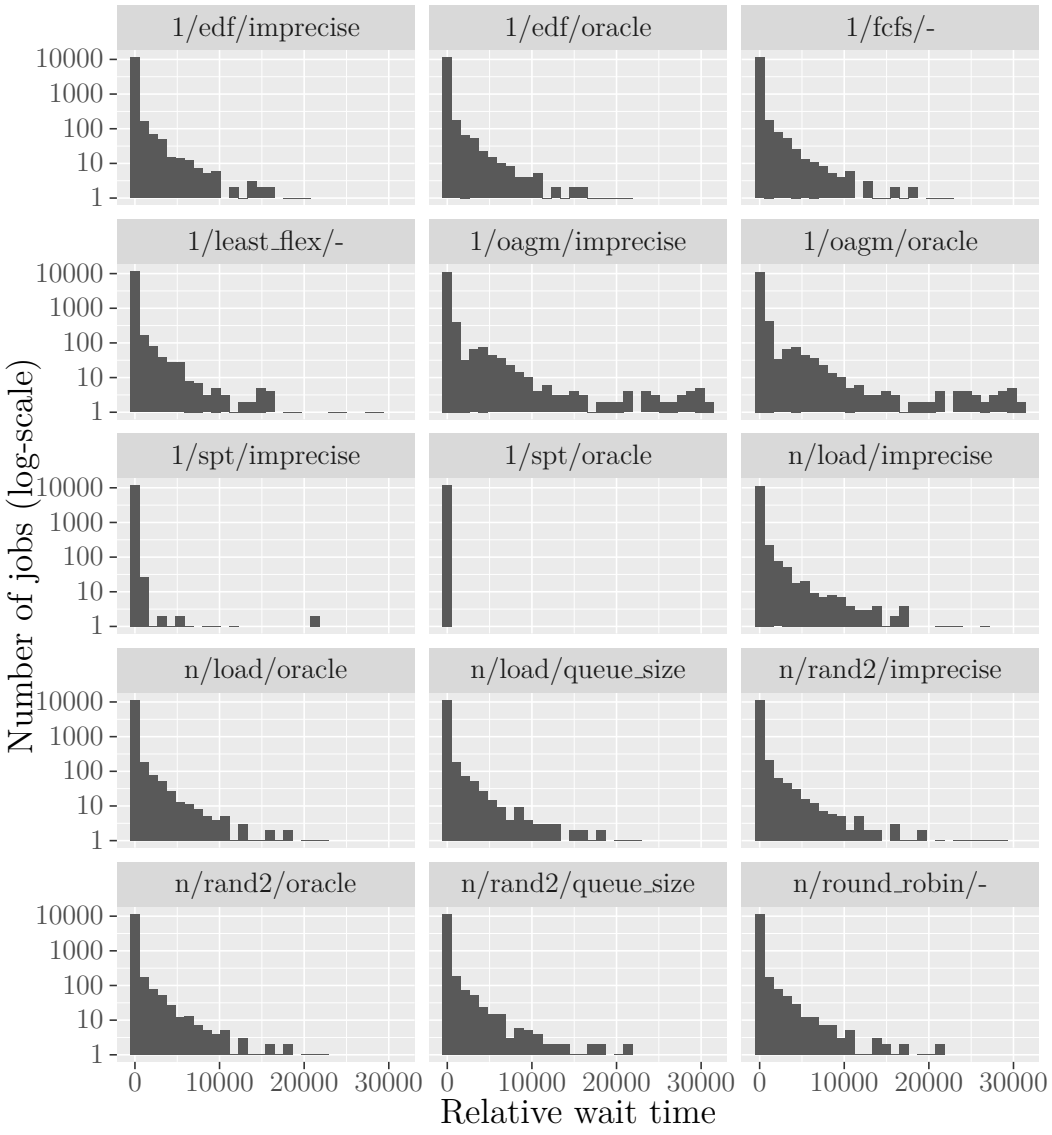

Figure 3.7: Histogram of relative wait times for each queue manager, throughout all executed workloads

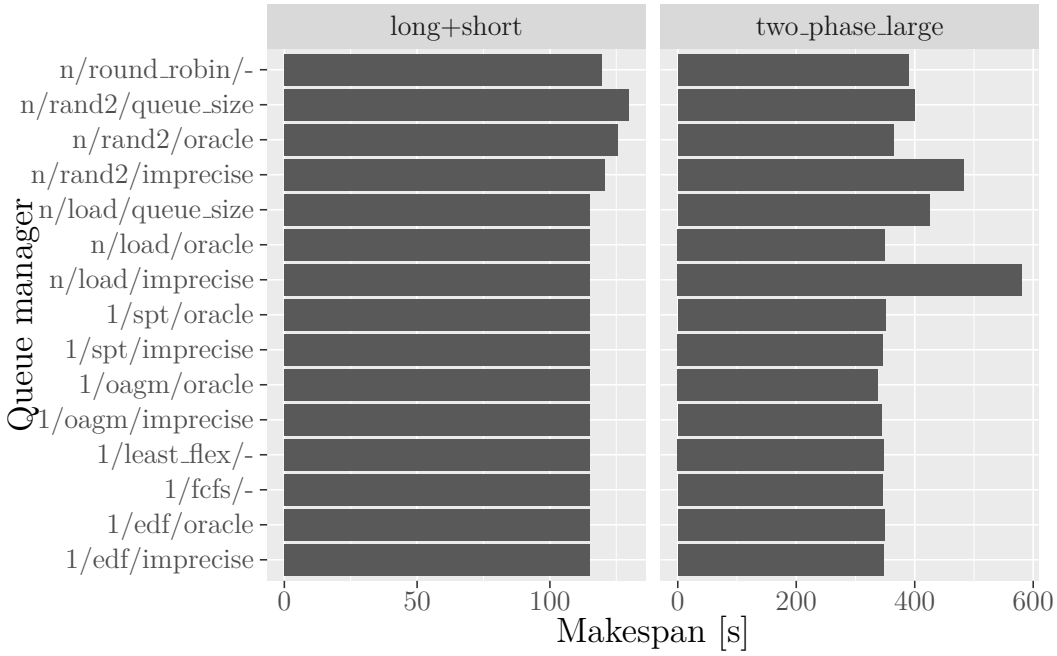

Figure 3.8: A comparison of makespans for individual queue managers and selected workloads

be improved by adjusting the deadline thresholds to fit the actual running times of jobs better, or by implementing a mechanism that adjusts them automatically.

We have also found that imprecise processing time estimation does not greatly affect the performance of single-queue load balancing algorithms, but it causes longer makespans in multi-queue algorithms. As a side note, small flexibility is a general problem of multi-queue algorithms that dispatch jobs immediately after arrival. In setups where workers can go offline at any time or where new workers can be added to the pool over time, additional measures must be taken to redistribute the work and avoid starvation.

4. Advanced Usage of Containers

In previous chapters, we only considered containers as isolated execution environments for programs that save computing resources thanks to sharing most of the operating system with the host machine (as opposed to virtual machines, for example). Docker containers have also been used as an execution unit for assignment evaluation on their own[38].

However, the capabilities of modern container platforms extend beyond this scope. They provide features such as automated building of containers and transferring them between physical hosts.

This particular functionality can be implemented in multiple ways:

- build containers as Docker images using Dockerfiles (a script-like format that describes the steps to build a container)
- use an alternative image builder based on Dockerfiles (such as `podman`[11] or `buildah`[12])
- script their creation using simple shell scripts
- use specialized tools such as Ansible or Chef.

Execution of programs in containers can be also done in many ways independent of the build mechanism – even Docker images can be unpacked and executed directly or using a different container runtime such as LXC or slurm.

In this chapter, we discuss the possibilities of exploiting these features to solve previously mentioned problems in systems for evaluation of programming assignments. We also implement a small part of the functionality to show that such solution is practically viable.

4.1 Analysis

The evaluation process in a system for programming assignment evaluation resembles the way a continuous integration service (a service that ensures the quality of a code base with each change, typically by building it and running unit tests or various static analyzers) works. Submitted source code is built using a preconfigured toolchain and then subjected to tests. Based on its performance, the solution is awarded a rating. The main difference from a classical continuous integration service is that performance and resource usage is a key element of the rating.

Modern continuous integration services often use containers for various tasks, and it is reasonable to presume we might be able to use them in a similar fashion since the problem of assignment evaluation is so similar to that of continuous integration.

4.1.1 Secure Execution

Public continuous integration servers run thousands of builds on completely untrusted code bases every day. Just as in our case, there are two basic scenarios to

address – intentional attacks and programming errors that lead to system failures.

It is sensible to consider different approaches to secure execution in ReCodEx than using `isolate`, since it causes problems with the stability of wall-clock time measurements (as shown in Chapter 2), and it affects the overall stability of measurements as well.

4.1.2 User-defined Runtime Environments

The steps required to build and test a code base can vary enormously. The number of existing programming languages, build toolkits, testing frameworks and third party libraries is so large that this process is almost unique for every project.

Continuous integration services need to provide a way for users to specify the building and testing process in a manner that is both simple and versatile enough to satisfy the needs of any project.

It seems that the situation is simpler for programming assignment evaluation systems. The assignments are usually much less complex than a typical project that requires continuous integration. Thanks to this, we do not usually need to set up complicated build pipelines for every new exercise. However, in some areas such as web development, it is common to use a large number of third-party libraries. Over time, this trend has also found its way into ReCodEx. For example, machine learning classes require a Python environment with the TensorFlow library installed, along with supplementary packages such as `gym`.

We can expect that in the near future, new exercise types with complex external dependencies will appear, and installing these dependencies directly on the host system will no longer be acceptable. There are two important arguments to support this prediction. First, some advanced libraries should not be available for evaluation of basic exercises. Second, backward incompatible changes are common in many community-maintained libraries and some exercise types might require incompatible dependency versions.

It is clear that we need a way to let exercise authors create a new, independent runtime environment based on an existing one, and that we can draw inspiration from continuous integration services, even though the build and testing steps we require are usually much simpler.

4.1.3 Preparation of Build Environments

As mentioned in previous sections, the requirements on installed build tools, libraries and other utilities can vary a lot with every project. The overhead of maintaining this manually on multiple worker machines would be hard to manage.

Moreover, in ReCodEx, adding support for a new build environment can mean not only installing the required tools on each worker machine, but also modifying the code of the HTTP API so that the frontend can use the new environment.

Since we aim to support user-defined software configurations, we should investigate how containers are used in existing continuous integration services to alleviate this maintenance overhead.

4.2 Related work

In Section 4.1, we listed requirements on a programming evaluation system that could be solved using container technologies. Because of the similarity between automated assignment evaluation and continuous integration, we shall survey a handful of public continuous integration services and the way they address these requirements.

4.2.1 GitLab CI

GitLab is a complex platform for developers based on the `git` version control system that can be both used as a service and hosted on private infrastructure. Continuous integration is one of its features beyond the scope of source code management.

The build configuration of a project is a YAML file that specifies a series of jobs to be executed. The actual commands to be executed in order to process a job are specified either by referencing a shell script or a Docker image name and a command to launch inside a container based on the image.

The jobs are executed with the help of GitLab runner – a program that can be installed on worker machines that process builds. Shell scripts invoked by jobs are executed without any security layer, which means allowing those is not suitable for publicly accessible runners. Docker-based jobs are considered safe enough for public runners. The runners shipped with the community GitLab instance are run in a virtualized environment[39], which provides another layer of security, along with the possibility of on-demand scaling.

In order to prepare a custom build environment, one must either configure a custom runner and use a script-based job, or build a Docker image, push it to a Docker registry and use it in the job specification. When the second way is used, the GitLab runner automatically fetches the image from the registry before the build.

With GitLab CI, it is also possible to use Docker containers to launch additional services needed for testing the project – a typical example would be a database or file storage server.

The way the building and testing process is described and processed in CircleCI is very similar to that of GitLab CI. Since the way our requirements are handled is nearly identical, too, we will not cover CircleCI any further.

4.2.2 Travis CI

Travis CI is a popular continuous integration server with a straightforward configuration language. Unlike GitLab CI and CircleCI, it does not prefer using containers to pre-build the testing environment (although it is technically possible). Instead, the developers (or the community, in some cases) implement the support for each language individually. For many languages, testing against multiple versions is supported, and this is achieved either with tools specific for each language (such as `phpenv` for PHP) or using the system package manager (this is the case for Python).

Since the build configuration can contain arbitrary commands, it is in theory possible to support any language by installing the required tools before the build

itself. However, this approach tends to prolong the builds unnecessarily.

The security of the build process is ensured by launching a separate virtual machine that runs Ubuntu (a popular GNU/Linux distribution) for each build of a project.

4.2.3 AppVeyor

At the time of its inception, the most interesting feature of AppVeyor was support for Windows builds (alongside Linux builds). Later, other services also got this feature. The builds are performed in virtual machines created on demand for each build.

The virtual machines are created from an image with a predefined set of pre-installed packages. If the build has additional dependencies, they must be installed manually (for example, using the NuGet package manager), similarly to the case of Travis CI.

In the basic configuration, the builds rely on virtualization for isolation. A new virtual machine is created for every build.

4.3 Implementation Analysis

Based on our analysis of the features of continuous integration systems, we propose a way of leveraging container platforms in automated evaluation of programming assignments, using ReCodEx as a model. We also provide a basic implementation of this functionality.

Since Docker is one of the most well-known container technologies and all the continuous integration services we surveyed support it in some way, we will use it in our implementation. However, the ideas we present here should be trivially transferable to any container platform based on the OCI[10] (the Open Containers Initiative) specification.

4.3.1 Docker Overview

Before we describe the implementation itself, we shall outline the basic features of Docker needed to understand the reasoning behind our design choices.

Docker is a software distribution platform that uses Linux containers to build and ship programs along with their dependencies and to allow deploying them on any host, without conflicting with installed software and other containers.

This functionality is enabled by the Docker daemon. The daemon is controlled through an HTTP API, which is even used by the command line application typically used to manage Docker containers. Since the API is usually accessible through a UNIX socket, it is even possible to have containers that control the Docker daemon.

The most important concept is an image, which is basically a snapshot of a minimal operating system that contains an application and its dependencies. An image is composed of layers – collections of files that we can imagine being laid on top of each other, making only the latest version of a file visible. This mechanism is useful for efficient updates and extension of images – image authors can easily add layers to install additional libraries, for example. Every image contains a

manifest - a collection of various metadata about the image, its contents and the intended usage. It is also possible to add user-defined labels that could be leveraged by higher layers of an assignment evaluation system.

Images are used for the creation of containers. These can be thought of as concrete instances of an image. A container is created by adding a writable layer on top of the layers of an image, which is a rather fast operation. Typically, there should only be a single process running inside a container. However, this rule is not enforced by the platform in any way – it is more of a design recommendation. Containers are often configured using environment variables passed from the host when they are created. It is also possible to bind network ports of a container to host ports and to mount parts of the host file system into a container.

Typically, images are built using a `Dockerfile` – a structured file that contains instructions on the build process. These are for example copying files from the build machine and running shell commands inside the container. Each instruction results in the creation of a new layer. The `Dockerfile` format provides a simple way of extending an existing image. Alternative ways of creating an image exist too, such as launching a shell inside a container, making changes to it manually and then copying the writable layer of the container and marking it as a new image layer (using the `commit` Docker command).

The layers of images are stored persistently on the file system of the Docker host. Multiple storage backends exist, but `overlay2` is the recommended and most widely used one. It is based on the OverlayFS file system module present in the Linux kernel, which allows mounting several directories on top of each other, where files are read from the highest layer that contains the requested file and writes are only made to the uppermost layer. Alternatives to `overlay2` are for example `devicemapper` or `btrfs` (which uses copy on write file system subvolumes, a feature of the btrfs filesystem in Linux).

To allow transferring container images between hosts, Docker defines the Registry API, which has a simple implementation that can be deployed using Docker. The Registry API was standardized by the OCI distribution specification after being used extensively in practice. The daemon can upload images to the registry with the `push` command and download them with the `pull` command. Both of these commands utilize layers, which makes transferring the images efficient.

4.3.2 Secure Execution

It is not surprising that virtual machines are considered secure enough to run untrusted code by the likes of Appveyor and Travis CI. The more interesting fact is that Docker containers are also used as an isolation layer (for example in CircleCI).

Since Docker uses Linux namespaces to create an isolated environment for the programs running inside the container, it should, at least in theory, be on par with `isolate`. In the default setup, it does not feature measurements and limiting of resource usage (time, memory, disk usage, ...). With further configuration, a memory limit can be set. However, adding time and disk usage limits and gathering statistics after the evaluated solution finishes would require implementing a supervisor program instead of just launching Docker containers and waiting for them to finish.

Because Docker cannot be used for securely measuring assignment solutions on its own, it will probably not replace `isolate` in ReCodEx in the near future. However, there are ways in which it could complement it – for example, it could be leveraged to add precise definitions of runtime environments – in ReCodEx, they all rely on the diligence of administrators to function correctly.

4.3.3 Deployment of New Runtime Environments

CircleCI and GitLab CI both run tests in Docker containers and it is possible to supply custom containers, which allows a great deal of flexibility and also a performance increase, since the environment does not have to be prepared again with each build.

This kind of functionality would also be useful in ReCodEx. An additional benefit in this case would be that the evaluations would be more reproducible, theoretically even after a longer period of time.

There are two possibilities of integrating containers into ReCodEx. We could put the worker binary itself into a container, along with `isolate` and the tools required by a particular set of runtime environments, to create an all-in-one image of sorts. This approach is similar to supplying custom runners in GitLab CI. The alternative is packaging each runtime environment in a separate image, keeping the worker separated and implementing launching images in the worker, which is more similar to how CircleCI and GitLab CI work with containers.

The all-in-one image approach allows things such as running specialized, heavily modified versions of the worker itself, which might be needed by some future exercises (for example running a program on a remotely controlled cluster of servers). Also, while adding a new runtime environment would mean extending the worker image and deploying it everywhere, we would not have to change the worker selection algorithm in the broker – the available runtime environments could still be enumerated in the configuration broadcast by the worker bundled in the image.

However, extending a particular environment with a library only for a set of exercises would either mean adding it to an existing environment (thus making it accessible in other assignments), or creating a new, globally visible runtime environment that would eventually be deployed on every worker. Supporting multiple versions of libraries or runtime environments at the same time would also be a challenge.

An advantage of the worker-less, single purpose image approach is that we would not need to maintain a large worker image with all the desired runtime environments. The images would be smaller, easier to review and faster to build. Also, a single worker could handle multiple versions of a runtime environment without significant effort, which would contribute to the repeatability of assignment evaluations. A drawback of this approach is that support for fetching and updating these single purpose images would have to be added to the worker daemon.

We could also implement a middle-ground approach that puts the worker inside a container (similarly to the all-in-one image variant), but only with the support for a single runtime environment (e.g., Java or Python). This would make it easy to create isolated, single-purpose runtime environment images. However,

every time a different environment was requested, the container with the worker would have to be stopped and a different image would have to be used. Instructing the host machine to switch worker containers would be a responsibility of the broker. The lack of advantages over the single purpose image approach and the need to add more responsibilities to the broker makes this approach impractical.

From the two basic approaches, we selected the second one, where only a single runtime environment is contained in each images and the worker is responsible for handling and switching the images. The main reason for this choice is that it makes it easier to maintain a set of curated runtime environments created by exercise authors. With the first approach, this would require administrators to modify the main worker image each time somebody wishes to create a new runtime environment. Furthermore, the second approach makes it much simpler to create single use environments that will only be used for a small set of exercises.

Building and Distribution of Images

In our implementation, we shall deploy a registry instance to be used by the workers and the Web API server, without being directly accessible to the public. The registry will serve as a storage of runtime environment images.

As opposed to the variant with a publicly accessible registry, we have full control of the images. However, we have to build them too, which would not be a requirement with the public registry – users could build the images locally and just upload them. Although it would be more convenient (we would not need to maintain a build service), it would also be much harder to audit the built images, since the images are basically just archives that can contain anything.

The exercise authors who desire to create a new runtime environment will simply write a `Dockerfile` that creates an image with all the tools they need to evaluate an exercise. The `Dockerfile` will then be uploaded to the backend using the Web API. After a review by an administrator, it will be passed to a build server, which will then push the resulting image to the registry, making it available to workers. An advantage of this approach is that it is easy to extend existing images using the `FROM` clause in a `Dockerfile`. Aside from being simple for exercise authors, this also allows for efficient use of storage space – the base image is only stored once and the images that extend it only contain new and changed files.

When a worker starts evaluating a solution that requires a newly created runtime environment, it will simply pull the image from the registry. The worker will also have to be modified to advertise the runtime environments that are available in the Docker registry so that the broker assigns jobs to it correctly.

4.3.4 Launching Containers with `isolate`

Each time the worker encounters a command that instructs it to execute a command in a container, it has to perform multiple steps:

1. Pull the image from the registry (if the image is not present yet)
2. Make the image accessible through the file system so that it can be used by `isolate`
3. Launch a command with `isolate`, using the image as the file system root

Of these steps, number 1 and 3 are rather simple. Number 1 only requires us to execute a single command and the Docker daemon will fetch all the files for us. Number 3 is already implemented by the worker and it does not need any adjustments.

Step number 2 is more challenging. A naïve approach would be to simply unpack the image data each time we need to execute a command in it. This is implemented by tools such as Charliecloud or `umoci`, but the process is not very complicated and we could also easily implement it on our own. Repeated copying of image contents is inefficient for obvious reasons – typically, an evaluation requires the execution of tens of commands and an image can have hundreds of megabytes. In total, this might cause a sizable overhead, as well as wear of the hard drive.

There are two similar ways we could improve this situation. First, we could mount the image exactly the way Docker does it. The default way is mounting the layers of the image using `overlay2`. Since the default configuration is also recommended by the developers and supported on all recent versions of the Linux kernel, supporting the other storage backends is not much of a concern.

A less complicated way of accessing the image contents is unpacking the images immediately at the moment when they are downloaded. Then, when the image is needed, we could simply link the image contents to the working directory of `isolate`. A drawback of this approach is that it is not as efficient with disk space usage. Each image has to be stored by the Docker daemon (which is efficient thanks to the layers mechanism), and the unpacked data have to be stored elsewhere, duplicating data of base images.

Since it is feasible to implement both of the alternatives and compare them, the exact method of making the image contents accessible to the evaluation sandbox should be chosen with respect to the results of this comparison.

4.3.5 Adding Auxiliary Services

Some types of exercises require a number of supporting services to run during the execution. For example, there might be an exercise in web programming that requires students to query a relational database server and output the result in a prescribed form. In this case, this requirement could be bypassed using a single file-backed database such as SQLite. However, this is not possible for every exercise type. There have even been exercises based on communication with a daemon-like service created by the exercise author.

The current job configuration format consumed by the worker does not allow this. With Docker, we can easily start a set of pre-configured containers using publicly available images (or, if needed, images from our private registry used for runtime environments) for each test and shut them down afterwards. These services will run in an isolated network namespace into which the tested program will be added. This ensures that we can run multiple evaluations of such exercises without a risk of network address and port conflicts.

Implementing this feature will require changes to the job configuration format. Since the individual tests of a solution should be isolated and independent on the order of execution, it would make sense to start the auxiliary services before each test and tear them down after it finishes. This could be implemented by

adding a parameter to execution types. However, cases exist where keeping the services running for multiple tests makes sense – for example, when the services are guaranteed to have no changing internal state or when they take a lot of time to launch.

Due to the explicit and rather verbose nature of the job configurations, we decided to create two new atomic task types that start and stop a container with a service. This will typically be hidden from end users by a higher level configuration form.

4.4 Implementation and Evaluation

To prove that the implementation we propose is practically feasible, we implemented three core parts of it:

1. fetching the contents of an image from a Docker registry,
2. unpacking an image into a directory by copying its layers, and
3. mounting the image layers using the `overlay2` file system driver.

The reason why we decided to also implement the image fetching is that it allows our solution to work on systems without a Docker installation. This is an important consideration because the Docker daemon is relatively complex and requires extensive privileges, which might be a concern for some administrators.

The unpacking and mounting of images are alternative approaches discussed in Section 4.3.4. We performed a rather simple experiment to compare their performance. We selected four official Docker images with varying overall size and number of layers:

- `alpine`, version 3.11.6 (2 MB, 1 layer)
- `fedora`, version 31 (64 MB, 1 layer)
- `python`, version 3.8.2 (341 MB, 9 layers)
- `glassfish`, version 4.1-jdk8 (334 MB, 10 layers)

For these images, we performed a sequence of 10 mount and unmount cycles and 10 unpack and remove cycles. This procedure was measured 100 times, giving 1000 measured operations in total. We measured blocks of 10 operations to alleviate the overhead of the benchmark program in cases where the operations only took milliseconds to complete (this is the case of the `alpine` image).

The measurements were performed on a laptop computer with an Intel i7-8850H CPU and a NVMe SSD hard drive (Toshiba KSG60ZMV256) running Linux 5.6.8.

The measurement results are depicted in Figure 4.1. We can safely conclude that unpacking an image by copying its content onto the file system is slower than mounting it, even though the difference is not as large as we have presumed. A more important result is that accessing image contents usually takes less than a second (and much less than that for small images), which is a reasonable amount of overhead in the case of programming assignment evaluation.

We decided that despite its large implementation complexity, directly mounting the layers of the image is a better approach because of its efficiency in terms of storage and because it is generally advisable to avoid writing large quantities of data on the hard drive frequently to prevent unnecessary shortening its lifespan.

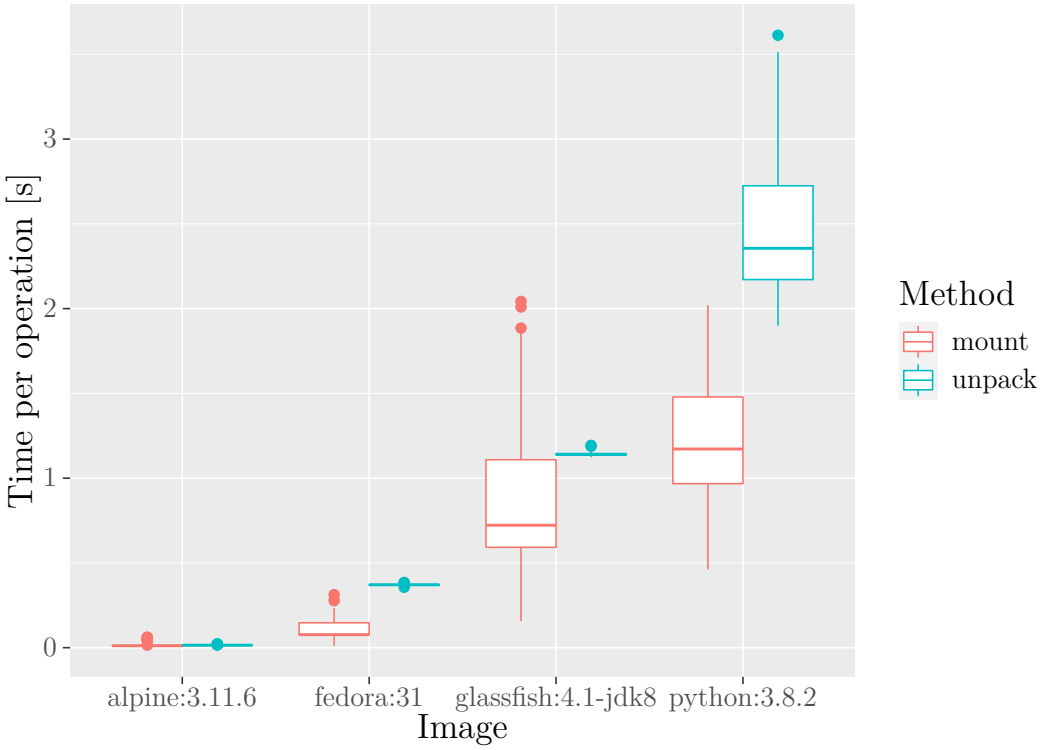

Figure 4.1: The results of experimental evaluation of the performance of unpacking and mounting the contents of various Docker images

4.5 Conclusion

We have shown that the proposed method of incorporating containers into programming assignment evaluation systems is practically usable. The parts we implemented as a part of this thesis can be easily adopted by ReCodEx or a similar system.

Employing containers helps solve the problem of scheduling over workers with diverse runtime environments by automating the process of distributing new or updated software to the worker machines, which leads to a more efficient operation and easier maintenance.

Using our work, it is also possible to add support for user-defined environments for specific courses. This helps widen the range of exercise types that can benefit from automated evaluation.

Even though our work is motivated by the Docker container engine, it does not depend on it directly. We do not require it to be installed on worker machines, runtime environments can be built with any tools that output OCI-compliant container images and any implementation of the OCI Distribution API specification can be used.

5. On-demand Scaling

In this chapter, we survey the current practical applications of on-demand scaling and their viability in automated evaluation of assignments. Although we do not contribute any implementation of this mechanism, we present an analysis and a set of design guidelines that can be built upon in the future.

On-demand scaling (or auto-scaling) is a feature of many virtualized machine platforms. While the exact mechanisms vary among different service kinds and providers, the basic principle remains. Execution units that are not utilized at the moment are returned to the provider, and when a traffic spike happens, the system can quickly react by allocating additional resources.

In this context, the allocated resources can be for example additional memory and CPUs (vertical scaling), or whole new virtual machines or containers (horizontal scaling). Due to our concern about measurement stability, we will only focus on horizontal scaling.

In a certain sense, on-demand scaling is also possible with physical servers – it is feasible to implement a resource management service that starts and shuts them down as necessary. Such approach might be even more efficient if we managed a large cluster of single-board computers.

The benefit of auto-scaling techniques is that they allow to save costs during low traffic periods, while being able to deal with unexpected increases in traffic when they happen. In systems for evaluation of programming assignments, the frequency of submissions can vary wildly, which means we might benefit from using these techniques. This is confirmed by submission data from ReCodEx, which shows that there is more traffic during evenings and that traffic decreases on Fridays and Saturdays (as shown by Figure 5.1).

5.1 Auto-scaling in Cloud Platforms

In this section, we compare the auto-scaling facilities provided by the three largest cloud providers as of today (based on anecdotal evidence) – Google Cloud Platform, Amazon Web Services and Microsoft Azure.

It seems that even though the product names and precise terminology differs, the mechanisms for auto-scaling offered by the three providers are mostly equal[40][41][42]. All the providers support rule-based horizontal scaling based on resource utilization over time. The supported utilization metrics include variations of CPU usage, memory usage and network traffic. All of the providers also allow the services to report custom utilization metrics that can be used for auto-scaling. The scaling rules typically instruct the platform to create or remove instances if the average of some utilization metrics over a time period reaches a user-defined threshold.

Another common feature is scaling based on time. If the user knows that the traffic is periodic, they can set up a rule that launches new instances at a given time of a day.

Amazon EC2 (a service included in AWS) also provides a service that does not seem to have an equivalent in the other platforms: Predictive Scaling[43]. It is advertised to use machine learning to predict traffic spikes and react to them.

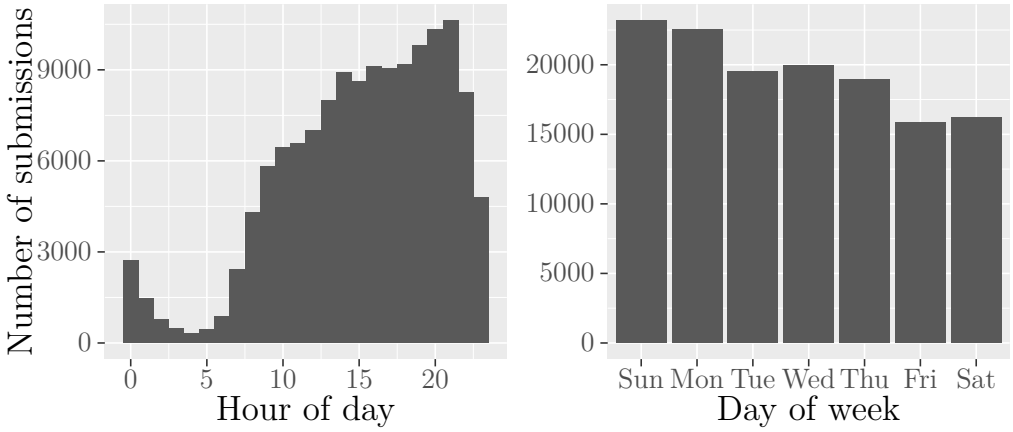

Figure 5.1: Number of submissions in ReCodEx, divided by hour and day of submission

This allows the end user to specify a utilization target (e.g., CPU utilization should be 80%) which will be maintained by the auto-scaler without a need to configure other thresholds.

5.2 Current State of the Art

Automatic scaling is a rather well researched problem with a large taxonomy of subproblems[44]. Many general and problem-specific approaches to auto-scaling exist an multiple frameworks have been proposed for the evaluation of their performance[45][46]. In this section, we present an overview of the problem and some of its details that are relevant to our efforts.

5.2.1 The Auto-scaling Process

The operation of an auto-scaling service (auto-scaler) can be decomposed into the following abstract steps[44]:

- **Monitoring**: A performance indicator metric is observed periodically.
- **Analysis**: The auto-scaler determines whether scaling actions should be performed based on the performance indicator data. Scaling can be performed either proactively (before a change in traffic happens) or reactively (after it happens). For successful proactive scaling, a prediction of future traffic is needed. Although more complicated, proactive scaling is a better approach when the scaling action takes a substantial amount of time. Also, oscillation (opposite scaling actions being done in a quick succession) should be mitigated at this step.
- **Planning**: The auto-scaler calculates how many resources should be allocated or deallocated to handle the traffic. It should also be aware of the budget imposed on cloud service usage.
- **Execution**: The actual execution of the scaling plan.

While the Monitoring step depends on the character of the workload and the Execution step depends on the execution model in place (such as virtual machines, containers or physical servers), the Analysis and Planning steps can be implemented in a more general fashion.

In the Analysis and Planning steps, various approaches are taken to process the performance data, such as:

- Rule-based policies
- Time-based policies
- Fuzzy rule-based policies (the rules do not have concrete parameters, their values are determined automatically)
- Predictions based on the rate of change (slope) of the performance data
- Regression models (linear regression, auto regressive moving average, etc.)
- Neural networks

Oscillation mitigation is typically implemented by either adjusting the thresholds dynamically or by adding a cooldown period after a scaling decision during which an opposite decision cannot be made.

5.2.2 Available Auto-scalers

The Kubernetes workload orchestrator is one of the leading drivers of auto-scaler development. One example of such auto-scalers is Clusterman[47], an auto-scaler developed by Yelp that also features a simulator that receives a time series of performance indicator values, and outputs whether the cluster is under-provisioned or over-provisioned and an approximate cost. As more examples, we can mention Cerebral (part of the Containership cloud platform[48]), KEDA[49] or Escalator[50].

ConPaaS[51] is an open source cloud platform that allegedly features automatic scaling. Unfortunately, we failed to find any documentation of this functionality so we cannot assess it.

5.3 Analysis

In previous sections, we have presented an overview of existing auto-scaling implementations. Here we assess the challenges of adapting them for automated evaluation of programming assignments. We focus on the Monitoring and Execution phases of the auto-scaling process, and also investigate the specifics of automatically scaling worker pools with different capabilities.

5.3.1 Execution Model

Automatic scaling is possible on many different execution models – using dedicated physical machines, virtual machines, containers and other abstractions.

Physical Machines

Remote management of server machines is a standard tool used by system administrators. Therefore it should be simple to launch and power off the machines automatically.

In the case of clusters of single-board computers, the situation is more complicated. Typically, these devices have a simple design without any facilities for being powered on remotely (such as Wake-on-LAN). This is justified by both the intended use case and the low overall power consumption. However, the device boots at the moment the power supply is plugged in, so it should be feasible to build a monitor machine that would manage the cluster by interrupting and restoring the power supply of each individual device. Another option is using a pre-built switched PDU (power distribution unit) for USB.

An obvious drawback of using physical machines owned by the maintainer of the system is that we cannot scale them up infinitely on demand – we are always limited by the amount of machines we have.

Virtual Machines and Containers

Virtual machines are a feasible execution model for assignment evaluation. Interfaces that allow automated launching and suspending of instances (such as REST APIs) are featured by most providers. However, using the services of common virtual machine providers requires caution. The most important consideration here is potential instability of measurements caused by interference with other virtual machines running on the same physical host.

There are two ways of resolution to this problem – either ignoring it, which might be valid for assignments that emphasize correctness of solutions and do not wish to grade them by efficiency, or using cloud platforms that can guarantee that our virtual machines do not share hardware with other tenants of the cloud platform. For example, Amazon AWS can provide such service, but it can be expected to incur additional costs.

As a side note, AWS also facilitates automated deployment of dedicated physical machines with remote access, which could be a viable option for auto-scaling physical servers.

As far as on-demand scaling is concerned, container platforms are very similar to virtual machine platforms. The problem with measurement stability is also present, and interfaces for automated management of instances (containers) are ubiquitous.

5.3.2 Performance Indicators

By performance indicators, we mean metrics that drive the decision to scale in or out. Our selection of the indicator is critical for the performance of the auto-scaling system.

CPU and Memory Utilization

In our case, The average CPU utilization over a certain time window does not indicate an overloaded system well. Even if this number is close to 100% for some worker, it can mean that it is processing a submission that takes a long time to evaluate and that it uses the CPU efficiently. Of course, it can also mean that the worker is continuously processing submissions and the queue is filling up, but we have no way to infer this from the CPU usage.

Low CPU utilization is a better marker in the sense that it is very probable that a worker for which this number is close to 0% is not very well utilized. There are cases where this might not be true, for example long-running IO-bound exercises, but it is a rare scenario in ReCodEx.

Memory utilization is not at all correlated with the actual utilization of a worker – it depends mostly on the submission that is being executed (and resource limits).

Network Traffic

In the case of ReCodEx, network traffic might be indicative of incoming jobs for a worker, because the workers communicate with the broker using TCP. However, there is no way to tell how long the queue is for a particular worker using this statistic on its own, similarly as in the case of CPU utilization.

Queue Length

As discussed in Chapter 3, we are able to estimate the length of a job on its arrival based on historical data. Using the same technique, it is straightforward to also estimate the load of a worker when we use a queue manager that maintains a separate queue for each worker.

For queue managers that use a single queue for all workers, we could calculate a potential load for each worker – the sum of estimated processing times of all the queued jobs the worker can process. Although this seems like a reasonable approach that is also utilized by the OAGM scheduling algorithm, an experiment that evaluates its usefulness would be necessary.

5.3.3 Load Balancing Constraints

In Chapter 3, we outlined a number of challenges specific to load balancing in a system for assignment evaluation. From these, only the need to handle arbitrary machine eligibility constraints is relevant to the auto-scaling problem.

The problem of auto-scaling workers with different capabilities is more difficult than when every worker can process any job. If a worker gets overloaded because it is the only one that can process some class of jobs, the auto-scaler needs to know what exact requirements the jobs have to launch a machine of the correct type and actually improve the situation. In the case of ReCodEx, this can be a substantial amount of information in the form of headers (key-value pairs with diverse semantics).

Interpreting arbitrary headers for efficient auto-scaling appears to be a difficult problem and we failed to find any prior art on this topic. This fact leads us to a conclusion that we should instead restrict the diversity of workers that are going to support auto-scaling. Since cloud auto-scaling typically allows defining multiple auto-scaling groups that are managed independently, we can require workers in each of these groups to have equal processing power and capabilities.

With this restriction in place, we cannot provide as much flexibility as we could if we supported arbitrary job requirements. However, it makes the work of the auto-scaler (and also the scheduler) much easier. If we employ container technologies to automatically deploy new runtime environments as laid out in

Chapter 4, we will also eliminate the only cause of diverse worker headers within a single hardware group encountered in ReCodEx until now.

5.3.4 Summary

Both physical machines and virtual machines are viable for usage in an automatically scaled system, but adjusting the total capacity of a virtual machine pool is a much simpler task than controlling physical machines. Being able to combine both execution models could also be valuable.

It is evident that low level metrics such as CPU utilization or network traffic do not reflect the actual worker utilization in a programming assignment evaluation system. The queue length, which is a more promising indicator, cannot be inferred by the auto-scaler itself from these low level metrics, and if we are to use it, we will have to implement support for reporting it to the auto-scaler from the broker.

Using arbitrary routing headers to select a machine for a job is likely to prove difficult to combine with on-demand scaling. Maintaining several scalable groups of workers with identical hardware and software configuration seems like a more viable approach.

5.4 Design Guidelines

As demonstrated by the previous sections, on-demand scaling is a rather complex problem that requires careful evaluation based on performance data from real traffic. We conclude that performing such an experiment (and implementing the related functionality) is out of the scope of this text, but we present a brief overview of how auto-scaling could be integrated into a programming assignment evaluation system.

5.4.1 Performance Monitoring

Since the queue length is the only useful metric of worker utilization, the load balancer (the broker in the case of ReCodEx) must be responsible for publishing performance data.

On most cloud platforms, this can be done either by invoking an HTTP API or using a command line program. This is the case for AWS[52] and Microsoft Azure[53] (which also provides an SDK for direct instrumentation of the monitored program). For the ReCodEx broker, using an HTTP API is certainly preferred over the CLI, since the infrastructure for working with HTTP requests is already implemented there.

For non-cloud use cases, leveraging a performance monitoring solution such as Prometheus[54] could be a possibility. For Prometheus in particular, a C++ client library exists that could be used to instrument the broker to send queue usage statistics that would be stored and processed by an auto-scaler.

5.4.2 Analysis and Planning

There are numerous approaches to the analysis of performance data and planning of scaling operations. Multiple solutions exist that could be used directly, such

as AWS Predictive Scaling or the auto-scalers mentioned in Section 5.2.2. It is worth noting that each of these would require a certain amount of experimenting with configuration parameters.

If a custom approach was chosen, it could either be implemented as a part of the load balancer (broker), which already has all the relevant information, or as a standalone service that would consume performance metrics from the load balancer or from a monitoring service such as Prometheus.

Adding a standalone auto-scaling service would not require any modifications to the ReCodEx broker since it already has the ability to deal with unexpected termination of workers and with new workers appearing. However, load balancing algorithms with immediate dispatch and no mechanism of redistribution should be avoided due to their inability to react to changes in the worker pool.

5.4.3 Execution

The method of execution of the scaling actions depends heavily on the exact computation model used in each deployment, such as:

- Spawning workers as virtual machines using a service such as Amazon AWS, Microsoft Azure or Google Cloud Platform, or a self-hosted solution like VMware vSphere.
- Running workers in containers orchestrated by Kubernetes or Apache Meson, for example.
- Turning physical servers on and off using a proprietary remote management console or KVM over IP
- Controlling a cluster of single board computers by switching their power supply on and off through a custom monitor device.

With the exception of the last one, all these methods are rather simple to implement, typically by leveraging HTTP APIs exposed by the respective services. This functionality should be included in the auto-scaling service (or the load balancer, if a monolithic architecture is preferred).

It is also possible to combine these methods as necessary to cover use cases such as using physical servers for assignments that require precise measurements and virtual machines for the other assignments.

5.4.4 Evaluation using Simulation

Using simulation to evaluate auto-scaling performance is preferred over measuring it on actual hardware or cloud platform to save costs and provide reproducible results.

If we were to use the simulator we implemented for evaluation of load balancing algorithms as described in Chapter 3, only minor changes would be required. New event types would have to be added for worker startup and shutdown, and for the invocation of the auto-scaler. The auto-scaler would then emit scaling actions that would result into worker startup and shutdown events being added to the simulation event queue.

To use the Clusterman simulator, a file containing performance metrics must be supplied as input. A random input generator has also been implemented as a part of the project. The simulator then outputs scaling actions.

6. Conclusion

We have presented and examined several problems related to large-scale deployments of programming assignment evaluation systems. Most of the results can be directly applied to the ReCodEx system, but they are general enough to also benefit other similar systems and even applications outside of the programming assigment evaluation domain where features like precise measurements or scheduling focused on delivering quick feedback for short jobs are desired.

Measuring the influence of running multiple measurements in parallel and using various isolation technologies on the stability of results provided us with insights about using modern CPUs for assignment evaluation. We found that simultaneous measurements interfere with each other, causing a decline in stability. A plausible explanation is that this is caused by concurrent usage of the memory controller and the last level of the CPU cache.

The most profound consequence of this is that it is not advisable to use CPUs with many cores and cloud platforms when stable measurements are required (e.g., performance-oriented programming courses or programming contests). We examined several methods that could alleviate this instability: explicit CPU and NUMA affinity settings and disabling logical cores. There is an improvement in stability when the available cores are split into disjoint groups, each of which is dedicated to a single evaluation worker, and when logical cores are disabled. This improvement, however, did not increase the number of parallel measurements that can be run simultaneously without a loss in precision on our testing machine.

Although we have shown that multi-core CPUs are not particularly suitable where precise measurements are required, they can be very useful for numerous other exercise types where returning the correct answer is more important than performance. It is a responsibility of administrators to allocate resources in a way suitable for the workload being processed by the system. However, the measurement framework we laid out can be a substantial aid in this task.

We also found that using isolation technologies affects the results of measurements, both in terms of overall speed and stability. Using the `isolate` sandbox seems to make the standard deviation of measurements higher than when they are performed on the bare metal or in Docker. Interestingly enough, this phenomenon was not as prominent when VirtualBox was used. Due to the nature of programming assignment evaluation, we cannot abandon isolation technologies. However, we should continuously evaluate their impact on measurement stability, which is one of the key elements of fair grading.

Our survey of online scheduling algorithms and a subsequent experimental examination yielded results that allow more efficient utilization of evaluation hardware. In addition to existing algorithms applicable to the problem, we have proposed two custom algorithms – one based on multi-level feedback queues and another based on the earliest-deadline-first approach. We have also contributed a practical implementation of the latter algorithm and included it in our experiment.

The experiment revealed that an algorithm that processes the job with the shortest processing time first has the best performance of the tested approaches in terms of the number of jobs processed without a large delay. A disadvantage

of this approach is that it requires a mechanism for estimation of processing time for incoming jobs. Fortunately, our results have shown that estimates based on historical data should be sufficient for this use case.

The performance of the custom earliest-deadline-first approach were very close to that of a trivial first come, first served algorithm. There is a possibility that this is caused by inadequately chosen parameters for determining job deadlines or by test inputs not being similar enough to real world jobs.

On-demand scaling of infrastructure is a topic that is related to load balancing. After a survey that explored the implementation possibilities ranging from physical server machines to containers and virtual machines provided by cloud computing platforms, we concluded that a practical implementation is out of the scope of this thesis. Nonetheless, we compiled a set of guidelines that could serve as a basis for future work.

During our research of scheduling algorithms, we found that container technologies could be used to simplify administration of job runtime environments over distributed evaluation workers. This would make it feasible to maintain a more homogeneous pool of workers where specialized software and updates can be deployed to all general purpose machines without manual intervention. Such improvement also makes scheduling more simple and more efficient.

We proposed a solution that automates the deployment of runtime environments, and also facilitates supporting custom environments defined by exercise authors (course instructors) without additional maintenance costs. We also implemented the core parts of this functionality and performed a simple experiment which showed that the overhead of the proposed solution is manageable (rarely over 1 second and typically in the order of hundreds of milliseconds for size-optimized environment images).

In summary, the presented results can serve as a foundation for building a large-scale system for evaluation of programming assignments that is efficient in terms of both cost and performance. This can help make programming education more efficient and accessible.

6.1 Future Work

Even though this thesis has fulfilled all of its objectives, there are multiple possible continuations of our research that could lead to even better performance in evaluation of programming assignments.

Single board computers seem like a promising platform for assignment evaluation. If we managed to prove that these devices can process assignments similarly to conventional computers and that their measurements are reasonably stable, they could be used for most exercise types. This has two considerable advantages. First, using each computer to only evaluate one submission at a time is not as wasteful as in the case of machines with modern multi-core CPUs. Secondly, the cost of adding a new single-board computer to the system is rather small in comparison to e.g., server machines. Due to these facts, we could increase the overall throughput of the system much more easily and without concerns about measurement stability.

Furthermore, implementing on demand scaling using switched power supplies could be a way to make the system more cost-effective while maintaining an

acceptable throughput during traffic peaks.

Another way of mitigating measurement instability introduced by isolation technologies and parallel measurements is repeating them multiple times. Proposing a process that summarizes these results for grading and evaluating it in various settings is an interesting research topic. If such process was found, it would allow us to use multi-core CPUs and cloud virtual machines more freely, without risking awarding a solution with an unfair grade due to measurement interference.

Additionally, the influence of this mechanism on scheduling should be studied, since the amount of work to be distributed over the worker machines would become larger. However, the processing times would become easier to predict if the implementation made the scheduler aware of the individual repeated measurements of each submission (as opposed to sending them as a bundle that performs all the measurements).

Finally, in our survey of scheduling algorithms, we did not closely examine those that employ preemption. Designing a preemption mechanism that could realistically be implemented in an assignment evaluation platform while keeping measurements stable would open a path to using more advanced scheduling algorithms that could prevent starvation of short jobs in a situation where the system is occupied by longer jobs.

Bibliography

[1] Martin Mareš and Bernard Blackham. "A New Contest Sandbox." In: *Olympiads in Informatics* 6 (2012).

[2] *chroot(2)*. 2019. URL: http://man7.org/linux/man-pages/man2/chroot.2.html (visited on 04/25/2019).

[3] Martin Mareš. "Perspectives on grading systems". In: *Olympiads in Informatics* (2007), p. 124.

[4] Stefano Maggiolo and Giovanni Mascellani. "Introducing CMS: A Contest Management System." In: *Olympiads in Informatics* 6 (2012).

[5] *Jails, FreeBSD Handbook*. URL: https://www.freebsd.org/doc/handbook/jails.html.

[6] *Security, FreeBSD Handbook*. URL: https://www.freebsd.org/doc/handbook/security-resourcelimits.html.

[7] *Commit adding namespaces to Linux*. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=071df104f808b8195c40643dcb4d060681742e29.

[8] *Notes from a Container*. URL: https://lwn.net/Articles/256389/.

[9] URL: https://www.docker.com/.

[10] URL: https://www.opencontainers.org/about.

[11] URL: https://podman.io.

[12] URL: https://buildah.io.

[13] URL: https://linuxcontainers.org/lxc/introduction/.

[14] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5 (2017).

[15] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged containers for user-defined software stacks in hpc". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–10.

[16] *OpenVZ Project*. URL: https://openvz.org.

[17] Rich Uhlig et al. "Intel virtualization technology". In: *Computer* 38.5 (2005), pp. 48–56.

[18] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177.

[19] *Kattis*. URL: https://www.kattis.com/.

[20] URL: https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt.

[21] *stress-ng*. URL: https://kernel.ubuntu.com/~cking/stress-ng/.

[22] Ole Tange. *GNU Parallel 2018*. Ole Tange, Mar. 2018. ISBN: 9781387509881. DOI: 10.5281/zenodo.1146014. URL: https://doi.org/10.5281/zenodo.1146014.

[23] Thomas J DiCiccio and Bradley Efron. “Bootstrap confidence intervals”. In: *Statistical science* (1996), pp. 189–212.

[24] Josh Aas. “Understanding the Linux 2.6.8.1 CPU scheduler”. In: *Retrieved Oct* 16 (2005), pp. 1–38.

[25] Kirk Pruhs, Jiri Sgall, and Eric Torng. *Online Scheduling.* 2004.

[26] Rajeev Motwani, Steven Phillips, and Eric Torng. “Nonclairvoyant scheduling”. In: *Theoretical computer science* 130.1 (1994), pp. 17–47.

[27] Kangbok Lee, Joseph Y-T Leung, and Michael L Pinedo. “Makespan minimization in online scheduling with machine eligibility”. In: *4OR* 8.4 (2010), pp. 331–364.

[28] S Anand et al. “Minimizing maximum (weighted) flow-time on related and unrelated machines”. In: *Algorithmica* 77.2 (2017), pp. 515–536.

[29] Fiona Fui-Hoon Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163.

[30] Shuang Cai et al. “Heuristic and Meta-heuristic Algorithms for the Online Scheduling on Unrelated Parallel Machines with Machine Eligibility Constraints”. In: *2018 14th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD).* IEEE. 2018, pp. 469–474.

[31] Yair Bartal et al. “On the value of preemption in scheduling”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques.* Springer, 2006, pp. 39–48.

[32] Nikhil Bansal et al. “Non-clairvoyant scheduling for minimizing mean slowdown”. In: *Algorithmica* 40.4 (2004), pp. 305–318.

[33] Luca Becchetti et al. “Semi-clairvoyant scheduling”. In: *Theoretical computer science* 324.2-3 (2004), pp. 325–335.

[34] Yossi Azar, Joseph Naor, and Raphael Rom. “The competitiveness of online assignments”. In: *Journal of Algorithms* 18.2 (1995), pp. 221–237.

[35] Jia Xu and Zhaohui Liu. “Online scheduling with equal processing times and machine eligibility constraints”. In: *Theoretical Computer Science* 572 (2015), pp. 58–65.

[36] Michael Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104.

[37] *ReCodEx Runtime Environments.* URL: `https://github.com/ReCodEx/wiki/wiki/Runtime-Environments`.

[38] František Špaček, Radomir Sohlich, and Tomas Dulik. “Docker as platform for assignments evaluation”. In: *Procedia Engineering* 100 (2015), pp. 1665–1671.

[39] URL: `https://about.gitlab.com/blog/2016/04/05/shared-runners/`.

[40] *Autoscaling groups of instances, Google Compute Engine Documentation.* URL: `https://cloud.google.com/compute/docs/autoscaler`.

[41] *Overview of autoscale, Microsoft Azure Documentation*. URL: https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-overview.

[42] *AWS Auto Scaling Features*. URL: https://aws.amazon.com/autoscaling/features/.

[43] *How Scaling Plans Work, AWS User Guide*. URL: https://docs.aws.amazon.com/autoscaling/plans/userguide/how-it-works.html.

[44] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. "Auto-scaling web applications in clouds: A taxonomy and survey". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–33.

[45] Alexey Ilyushkin et al. "An experimental performance evaluation of autoscaling policies for complex workflows". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 75–86.

[46] Alexandros Evangelidis, David Parker, and Rami Bahsoon. "Performance modelling and verification of cloud-based auto-scaling policies". In: *Future Generation Computer Systems* 87 (2018), pp. 629–638.

[47] URL: https://clusterman.readthedocs.io/.

[48] URL: https://containership.io.

[49] URL: https://keda.sh/.

[50] URL: https://blog.developer.atlassian.com/introducing-escalator/.

[51] Guillaume Pierre and Corina Stratan. "ConPaaS: a platform for hosting elastic cloud applications". In: *IEEE Internet Computing* 16.5 (2012), pp. 88–92.

[52] URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/publishingMetrics.html.

[53] URL: https://docs.microsoft.com/cs-cz/azure/azure-monitor/platform/metrics-custom-overview.

[54] URL: https://prometheus.io/.

List of Figures

List of Tables

Attachments

The following is a description of files contained in the attachments. The files are laid out as follows:

- `/measurement_stability` – scripts for evaluation of measurement stability
- `/measurement_stability/distribute_workers.sh` – a script that implements the placement of measurements on CPU cores as described in Section 2.2
- `/measurement_stability/workloads` – the programs measured in the experiments and test inputs
- `/measurement_stability/plots` – scripts for processing and plotting results
- `/scheduling` – implementation of various scheduling algorithms and scripts for their experimental evaluation
- `/scheduling/simulator.cpp` – source code of the simulator used for evaluation of queue managers
- `/scheduling/setups` – descriptions of simulated worker pools used in evaluation of queue managers
- `/scheduling/queue_managers` – source code of the evaluated queue managers
- `/scheduling/workloads/generators` – generators of random inputs for the queue manager simulator
- `/scheduling/plots` – scripts for processing and plotting of measurement results
- `/containers` – implementation of downloading, unpacking and mounting of OCI images
- `/containers/plots` – evaluation and plotting of results of performance measurements
- `/user_behavior` – scripts for collection and evaluation of data about user behavior
- `/user_behavior/predict_processing_times.py` – an implementation of the processing time estimation formula described in Section 3.4
- `/thesis.pdf` – an electronic version of this thesis

This description is not exhaustive. Each of the top-level folders mentioned here contain a `README.md` file with a detailed description of the contents and usage instructions.

The contents of the thesis and its attachments (excluding measurement results) can be found in a git repository at `https://github.com/Teyras/MT`.

A. Error of Isolate Measurements

Table A.1: Characteristics of the error of isolate CPU time measurements, sorted by the relative error (truncated)

Setup	Isolation	Workload	Mean error[s]	Rel. error[%]
parallel-homogenous,10	D+I	insertion_sort	0.003	0.235
parallel-homogenous,4	D+I	insertion_sort	0.003	0.233
parallel-synth-cpu,8	D+I	exp_float	0.003	0.232
parallel-homogenous,8	D+I	insertion_sort	0.003	0.231
parallel-homogenous,20	D+I	exp_float	0.003	0.226
parallel-synth-memcpy,2	V+I	exp_float	0.001	0.226
parallel-homogenous,6	D+I	insertion_sort	0.003	0.225
parallel-homogenous,2	D+I	gray2bin	0.003	0.224
parallel-synth-cpu,4	D+I	insertion_sort	0.003	0.223
parallel-synth-memcpy,20	D+I	insertion_sort	0.004	0.221
parallel-homogenous,10	I	insertion_sort	0.003	0.221
parallel-homogenous,6	I	insertion_sort	0.003	0.219
parallel-homogenous,8	I	insertion_sort	0.003	0.219
parallel-homogenous,4	I	insertion_sort	0.003	0.214
parallel-homogenous,40	D+I	bsearch	0.003	0.211
parallel-synth-memcpy,2	V+I	insertion_sort	0.001	0.208
parallel-synth-memcpy,4	D+I	gray2bin	0.003	0.204
parallel-homogenous,4	D+I	gray2bin	0.004	0.202
parallel-homogenous,10	D+I	gray2bin	0.004	0.201
parallel-synth-cpu,6	D+I	insertion_sort	0.003	0.197
parallel-homogenous,8	D+I	gray2bin	0.004	0.196
single,1	I	insertion_sort	0.003	0.196
parallel-synth-cpu,10	I	insertion_sort	0.003	0.195
parallel-homogenous,6	D+I	gray2bin	0.004	0.193
parallel-homogenous,8	I	gray2bin	0.004	0.192
parallel-homogenous,8	D+I	exp_float	0.003	0.192
parallel-synth-memcpy,2	D+I	qsort.py	0.039	0.191
parallel-homogenous,2	I	insertion_sort	0.003	0.191
parallel-synth-memcpy,2	V+I	gray2bin	0.001	0.191
parallel-synth-cpu,20	V+I	qsort.py	0.027	0.190
parallel-homogenous,10	I	gray2bin	0.004	0.190
parallel-synth-cpu,40	D+I	qsort.py	0.037	0.189
parallel-homogenous,2	D+I	insertion_sort	0.003	0.188
parallel-homogenous,4	D+I	exp_float	0.003	0.188
parallel-homogenous,10	D+I	exp_float	0.003	0.187
parallel-homogenous,4	I	exp_float	0.003	0.186
parallel-homogenous,6	I	gray2bin	0.004	0.186

Setup	Isolation	Workload	Mean error[s]	Rel. error[%]
parallel-homogenous,6	V+I	qsort.py	0.023	0.185
parallel-homogenous,4	I	gray2bin	0.004	0.185
parallel-homogenous,10	I	exp_float	0.003	0.184
parallel-synth-memcpy,2	V+I	qsort.py	0.019	0.183
parallel-synth-memcpy,6	D+I	insertion_sort	0.003	0.183
parallel-homogenous,10	D+I	exp_double	0.003	0.182
parallel-synth-memcpy,20	V+I	exp_double	0.004	0.182
parallel-homogenous,20	D+I	qsort	0.004	0.181
parallel-homogenous,6	D+I	exp_float	0.003	0.181
parallel-synth-cpu,2	V+I	qsort.py	0.020	0.181
parallel-homogenous,2	V+I	insertion_sort	0.001	0.181
parallel-homogenous,6	D+I	exp_double	0.003	0.180
parallel-homogenous,8	D+I	exp_double	0.003	0.179
parallel-synth-cpu,20	I	insertion_sort	0.002	0.178
parallel-synth-cpu,6	I	insertion_sort	0.003	0.178
parallel-synth-memcpy,10	I	insertion_sort	0.004	0.178
parallel-homogenous,8	I	exp_float	0.003	0.177
parallel-homogenous,6	I	exp_float	0.003	0.177
single,1	V+I	qsort.py	0.019	0.174
single,1	V+I	insertion_sort	0.001	0.174
parallel-synth-cpu,4	D+I	qsort.py	0.040	0.173
parallel-homogenous,10	I	exp_double	0.003	0.173
parallel-synth-memcpy,4	D+I	insertion_sort	0.003	0.173
parallel-homogenous,4	D+I	exp_double	0.003	0.173
parallel-homogenous,6	I	exp_double	0.003	0.172
parallel-synth-memcpy,40	D+I	insertion_sort	0.006	0.170

Table A.2: Characteristics of the error of isolate wall-clock time measurements, ordered by the relative error (truncated)

Setup	Isolation	Workload	Mean error[s]	Rel. error[%]
parallel-homogenous,40	D+I	insertion_sort	0.382	196.845
parallel-homogenous,40	D+I	exp_float	0.382	159.590
parallel-homogenous,40	I	insertion_sort	0.285	154.053
parallel-homogenous,20	D+I	bsearch	0.151	37.179
parallel-homogenous,40	D+I	qsort.py	0.287	29.900
parallel-homogenous,40	D+I	qsort_java.sh	0.428	23.014
parallel-homogenous,4	I	exp_float	0.049	9.555
parallel-homogenous,4	I	exp_double	0.051	9.317
parallel-homogenous,8	D+I	qsort_java.sh	0.198	4.954
parallel-homogenous,10	I	qsort.py	0.098	4.821
parallel-synth-cpu,40	I	exp_double	0.025	2.255
parallel-synth-cpu,8	D+I	qsort_java.sh	0.195	2.054
parallel-synth-cpu,10	I	gray2bin	0.032	1.816
parallel-synth-memcpy,4	D+I	qsort	0.031	1.660

Setup	Isolation	Workload	Mean error[s]	Rel. error[%]
parallel-synth-cpu,8	I	qsort	0.035	1.376
parallel-synth-memcpy,6	D+I	qsort	0.027	1.371
parallel-synth-memcpy,8	I	bsearch	0.028	1.362
parallel-synth-memcpy,10	D+I	bsearch	0.026	1.360
parallel-synth-cpu,20	D+I	bsearch	0.020	1.187
parallel-synth-memcpy,20	D+I	bsearch	0.027	1.028
parallel-synth-cpu,20	I	bsearch	0.019	1.012
parallel-synth-memcpy,2	I	exp_double	0.021	1.005
parallel-synth-cpu,4	V+I	qsort_java.sh	0.074	0.807
parallel-synth-memcpy,8	D+I	qsort.py	0.064	0.799
single,1	I	insertion_sort	0.021	0.795
single,1	I	gray2bin	0.021	0.737
parallel-synth-memcpy,2	I	bsearch	0.022	0.716
parallel-synth-cpu,2	I	bsearch	0.022	0.702
parallel-synth-cpu,8	V+I	insertion_sort	0.001	0.252
parallel-synth-memcpy,2	V+I	gray2bin	0.002	0.249
parallel-synth-cpu,20	V+I	qsort.py	0.029	0.213
parallel-homogenous,4	V+I	insertion_sort	0.001	0.174
parallel-synth-memcpy,4	V+I	insertion_sort	0.001	0.171
parallel-homogenous,8	V+I	insertion_sort	0.002	0.169
parallel-synth-cpu,6	V+I	exp_float	0.001	0.134
parallel-synth-memcpy,8	V+I	gray2bin	0.002	0.132
parallel-homogenous,8	V+I	qsort.py	0.026	0.123
parallel-synth-cpu,2	V+I	bsearch	0.001	0.121