Smart Resource Management for Data Streaming using an Online Bin-packing Strategy

Oliver Stein*, Ben Blamey†,Johan Karlsson‡, Alan Sabirsh‡, Ola Spjuth§, Andreas Hellander†, and Salman Toor†
* Department of Pharmaceutical Biosciences, Uppsala University, Sweden
Email: {Oliver.Stein, Ola.Spjuth}@farmbio.uu.se
† Department of Information Technology, Division of Scientific Computing, Uppsala University, Sweden
Email: {Ben.Blamey, Andreas.Hellander, Salman.Toor}@it.uu.se
‡ Discovery Sciences, Innovative Medicines, AstraZeneca, Gothenburg, Sweden
Email: {Johan.Karlsson1,Alan.Sabrish}@astrazeneca.com

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378241

***Abstract*—Data stream processing frameworks provide reliable and efficient mechanisms for executing complex workflows over large datasets. A common challenge for the majority of currently available streaming frameworks is efficient utilization of resources. Most frameworks use static or semi-static settings for resource utilization that work well for established use cases but lead to marginal improvements for unseen scenarios. Another pressing issue is the efficient processing of large individual objects such as images and matrices typical for scientific datasets. HarmonicIO has proven to be a good solution for streams of relatively large individual objects, as demonstrated in a benchmark comparison with the Apache Spark and Kafka streaming frameworks. We here present an extension of the HarmonicIO framework based on the online bin-packing algorithm. The main focus is to compare different strategies adapted in streaming frameworks for efficient resource utilization. Based on a real world use case from large-scale microscopy pipelines, we compare two different strategies of auto-scaling implemented in the HarmonicIO and Spark Streaming frameworks.**

***Index Terms*—Data Streaming, Resource Management, Cloud Infrastructures, Scheduling, Big Data, Scientific Data analysis, Online Bin-packing, Profiling**

I. Introduction

Production-grade data stream processing frameworks such as Spark[1], Kafka[2] and Storm[3] have enabled efficient, complex analysis on large datasets. These frameworks feature reliable transfer of the data, efficient execution based on multiple processing units, in- or out-of-order processing, and recovery from failures. These features are fundamental to develop production-grade streaming applications, but are not themselves sufficient to guarantee efficient utilization of resources. Indeed, with the popularity of public cloud infrastructures based on a pay-as-you-go model, the extended list of requirements both for the streaming frameworks and for the applications that run using these frameworks include efficient utilization of resources to reduce the cost of running applications, and rapid deployment of frameworks on different platforms. To achieve this, streaming frameworks need to be resource-aware in order to achieve the best possible resource utilization based on different scaling mechanisms.

Moreover, frameworks need to be flexible enough in terms of management of inhomogeneous compute and storage resources since this allows for scaling processing units based on the best possible prices available on the public platforms. On the application side, one requirement is to design self-contained applications that can be deployed seamlessly on a variety of resources. To that end, different efforts have been made. The most popular self-contained application design scheme is the containerization approach. Based on Docker[4], LXD[5] or Singularity[6] container software, applications can easily be deployed on a variety of resources.

For intelligent resource management, different machine learning approaches both from supervised and unsupervised learning have been extensively studied [1]. However, it has been observed that in order to make supervised learning approaches effective, regular re-training is required to cope with evolving scenarios [2], [3]. Unsupervised learning on the other hand needs longer time to provide reasonable estimates [4]. Additionally, a large quantity of the published work in this area is based on synthetic datasets where real environment challenges are not very well covered.

A challenge often overlooked is the efficient processing of large individual objects in an online stream-processing setting. Most of the currently available streaming frameworks (and benchmarking studies) focus on processing very large datasets composed of many small individual data objects [5], [6], such as XML or JSON documents representing social media data, e-commerce or financial transactions, web or mobile analytics, or server logs. In these applications, the size of each individual object typically ranges from bytes to kilobytes. Conversely, for typical scientific datasets composed often composed of large matrices or images (from e.g. microscopy), the individual objects are often relatively large (ranging from mega- to gigabytes). The support for handling online streams of such

[1]https://spark.apache.org/
[2]https://kafka.apache.org/
[3]https://storm.apache.org/
[4]https://www.docker.com/
[5]https://linuxcontainers.org/
[6]https://singularity.lbl.gov/

data is lacking.

Motivated by applications in large-scale processing of microscopy images, we recently developed HarmonicIO (HIO), a data streaming framework geared towards efficient processing of streams composed of large individual objects [7]. Recent performance benchmark comparisons of HarmonicIO with the Spark and Kafka streaming frameworks illustrate the increased throughput that can be expected from HIO in that scenario [5]. The architecture and the features of HarmonicIO are further discussed in Section III.

In this article, we present an extension of the HIO framework with features for context-aware and efficient resource utilization by introducing an Intelligent Resource Manager (IRM) component. Our approach is based on dynamic online bin-packing, a lightweight and extremely efficient algorithm that lets us optimally utilize available compute resources. In contrast to solutions based on machine learning, the online bin-packing algorithm does not require training data and model fitting. Instead, we employ a run-time learning process that profiles the characteristics of the running workloads. The proposed IRM extension thus relies on dynamic online bin-packing in order to schedule the workloads based on their run-time resource usage.

To test the IRM extension we evaluate the system in an environment hosted in the SNIC science cloud, with tests based on both synthetic and use-case based workloads. We show a high degree of efficiency in resource scheduling from the bin-packing algorithm, and we demonstrate how the system is able to match the scheduled CPU usage and actual usage accurately in a real environment.

Specifically, we make the following key contributions:

- Present an extension of the HarmonicIO framework with efficient resource utilization using dynamic online bin-packing algorithm.
- Provide an extensive evaluation of the proposed IRM component.
- Thoroughly compare the here proposed resource allocation strategy with the rule based strategy used in Spark Streaming for a real-world scientific workload.

The article highlights the underlying challenges and proposes a solution for efficient resource utilization in the highly dynamic environment of streaming frameworks. We have highlighted the two different strategies of auto-scaling used in HarmonicIO and in the latest version of the Spark Streaming framework. HarmonicIO utilizes dynamic online bin-packing whereas Spark's approach is based on rule-based fixed allocation strategy for different workloads. Our experiments are based on real scientific environment settings and the presented comparison with Spark shows the value of the proposed solution and the strength in the proposed resource allocation strategy.

The remainder of the article is organized as follows. Section II reviews state-of-the-art approaches for the efficient resource utilization in streaming frameworks. Section III explains the architecture and the features of HarmonicIO. The online bin-packing algorithm is covered in Section IV. Section V explains the integration details of the proposed IRM component and the HarmonicIO framework. Results are presented in the Section VI and Section VII summarize the article and outlines future research directions.

II. Related Work

Various approaches have been explored to address the challenge of efficient resource utilization. For example, a popular domain has been control theory, with previous work investigating how to use Kalman filters to minimize operational resource costs [8], or to track CPU usage and accordingly update resource allocations for workloads as they vary [9]. The interesting feature of Kalman filters is the predictive estimations of future behaviour, allowing the workloads to be captured increasingly accurately. The difficulty in applying the filters to resource scheduling lies in modeling the cost functions to minimize and the control system, in order to achieve their full potential. In contrast to these works that targets bare-metal and VM environments, the solution proposed in this article targets resource scheduling based on containers under a data streaming setting. The main goal of adaptive resource utilization optimization remains the same.

Another interesting approach is to use overbooking, as proposed by [10]. They designed a model based on overbooking combined with risk assessment to maintain tolerable performance levels and at the same time keep a minimum level of resource utilization across multiple resources (CPU, memory, network I/O etc.). This reduced overestimation of resource requirements and helped server application collocation. In comparison our approach assigns the scheduling of computing resources to the streaming framework rather than the user having to provide information about the workloads.

Bin-packing has previously been used for scheduling workloads in cloud computing contexts. In [11], a resource manager for cloud centres was proposed, featuring dynamic application reallocation with bin-packing based on run-time workloads. With the help of the resource manager, the number of VMs required to host the applications could be reduced. Based on these promising results, the work on our proposal was inspired by the use of bin-packing for optimizing resource utilization. However, we opted to use the bin-packing on a container level, gearing towards the very popular containerized approach today.

Furthermore, reinforcement learning (RL) is another appealing domain for exploring optimal auto-scaling policies. The methods rely on an exploration and exploitation approach. The idea is to get a reward or penalty based on the decision and a policy will be designed by maximizing the rewards. The method works very well in many dynamic systems. However, the challenge is to calibrate a trade-off between exploration and exploitation. With a strict exploitation approach the system will be reluctant to try new polices. On the other hand, too much exploration leads to longer time to set a policy. The paper [12] discusses the marginal performance of the Q-learning RL method for auto-scaling scenarios. Furthermore, papers [13], [14] present advanced RL approaches for auto-scaling scenarios. Our proposed approach based on bin-packing is not

limited by the incentive based strategies, yet it is still flexible enough to adapt according to the dynamic requirements.

III. The HarmonicIO Streaming Framework

HASTE (Hierarchical Analysis of Spatial and Temporal Data) is a research project funded by the Swedish Foundation for Strategic Research (SSF), proposing a "*hierarchical approach to acquisition, analysis, and interpretation of image data*" [15]. A central future goal is to develop intelligent workflows that allows for dynamic and online re-configuration and data prioritization for large-scale and long-running imaging experiments. For that, high-throughput stream analysis of images will be necessary.

One of the components of the HASTE platform is the stream processing framework HarmonicIO, introduced in [7]. HIO uses Docker containers as *processing engines* (PEs) to process the streamed messages; these are designed and provided by the client based on a template. A deployment of the framework consists of a single master node, and several worker nodes hosting PE containers.

To work with HIO, a user will first need to design their desired data processing task as a Docker container image and publish this to Docker Hub; examples and a template are available on the HASTE GitHub repository[7]. With this in place the user can start querying a HIO server to host PEs based on this container image and start streaming the data to process, using the *HarmonicIO Stream Connector*, a Python client API. From here HIO takes care of directing the data streams to the intended PE endpoint. Figure 1 is an illustration of the architecture. A key feature is smart P2P behaviour, where messages are sent available PEs for processing directly from the client, falling back to a backlog queue on the master node as needed.

One of the strengths of HIO lies in the throughput for larger object sizes. Since one of the goals of the HASTE project is to analyze microscopy images, the target object size is much different to the typical workloads often consisting of streaming text files or JSON objects. [5] compared HIO to similar common streaming frameworks, namely Apache Spark Streaming and Kafka, and found that HIO could achieve higher throughput for larger object sizes, under some configurations.

A. Architecture

HIO's peer-to-peer (P2P) architecture allows messages to be transferred directly from source nodes to PEs for processing, falling back to a queue at the master node if if message ingress rate exceeds processing capacity. Messages in this queue are processed with higher priority than new messages.

HarmonicIO has the following overall architecture (see Figure 1):

- **Stream connector** The stream connector acts as the client to the HIO platform, handling communication with the REST APIs of the other nodes, so that the user can stream a message. Internally, it requests the address of an available PE, so the message can be sent directly if possible. A stream request message consists of both the data to be processed, and the Docker container image and tag that must be running in the PE to process the data.
- **Master** The master node of HIO is responsible for maintaining the state of the system, tracking worker nodes and the availability of their containers, connecting stream requests to workers that are available and starting PEs as per user requests. It also maintains a backlog queue of streaming messages, if the message ingress rate exceeds available processing capacity.
- **Worker** The workers host PEs which contain the user's code to process data that is streamed to the PE via P2P from the stream connectors. The workers nodes report to the Master node, and can be seen as a pool or set of available streaming endpoints.

[7]https://github.com/HASTE-project/HarmonicPE

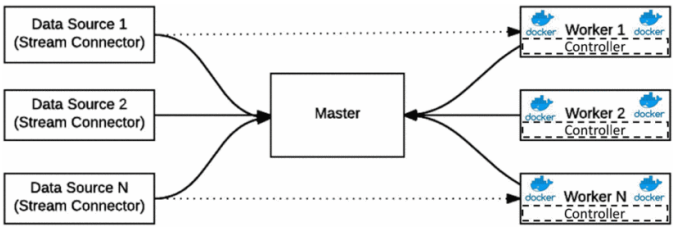


Fig. 1: The HarmonicIO architecture. The system consists of a master node, worker nodes and stream connectors, where solid lines indicate communication and dotted lines P2P data transfer. Image from [7, Fig. 1].

B. Extending HIO with the IRM

In [16], several changes and additions were made to the HIO architecture in order to add support for dynamic workload scheduling and improve resource management. The main contribution was to add the possibility for the master node to autonomously decide where to host containers based on an algorithm, such as bin-packing, and to allocate PEs when needed, as well as adding CPU profiling of processing engines during run-time on the workers. Previously, the user had to manually request a specific worker to host a PE; this is still possible but no longer necessary.

The outcome of the work led to the extension of HIO proposed in this article which adds intelligent resource management based on bin-packing, presented in more detail in Section V.

IV. Online Bin-packing

In [17] *bin-packing* algorithms are described as algorithms aimed at solving the optimization problem of assigning a sequence of items of fixed *size* into a number of *bins* using as few of these as possible. Furthermore, [18] describes *online* bin-packing as the case where each item in the input sequence is assigned one by one without knowledge about the following items, meaning that information about future items is not contributing to the placement. [17] also mentions

the *asymptotic performance ratio*, denoted R, which indicates the number of bins an algorithm needs as a factor of the number of bins in the optimal solution. Denoting the optimal number as O, online bin-packing algorithms will thus use $R \times O$ containers. Multiple studies [19]–[22] have analyzed the performance of these algorithms, and generally they perform well when comparing the cheap cost in time and memory to the approximation results.

A. The First-Fit algorithm

Several online bin-packing algorithms were studied in [19]. In particular, a group of such algorithms were examined, namely the Any-Fit group. Relatively simple, they share a common approach for finding the bins in which to put the next item, and the best performance ratio in the group is proven to be $R = 1.7$. The common approach is detailed in Algorithm 1, where the input is a sequence of items, $L = (a_1, a_2, \ldots, a_n),\ a_i \in (0, 1]$. The items are packed in order and a_i corresponds to the item size. The list of currently active bins is denoted as $B = (b_1, b_2, \ldots, b_m)$, and m is the number of bins needed at the end of the algorithm. As indicated, new bins are only generated when no currently active bin can fit the next item.

Of particular interest is the First-Fit algorithm, with a ratio of $R = 1.7$ as well as $O(n \log n)$-time and $O(n)$-space complexities, and is the algorithm that we based our resource management optimization upon. The search criterion in First-Fit is to find the first (lowest index) available bin in the list in which the current item fits.

```
for i := 1 to n do
    begin
        find available bin b_a in B according to criterion
        if a_i fits in b_a then
            place a_i in bin b_a
        else
            allocate new bin b_new and add to B
            place a_i in bin b_new
        end
    end
end
```

Algorithm 1: General Any-Fit approach

B. Dynamic bin-packing

Another extension of the problem is when the items modeled in the bin-packing algorithm can be removed from the bins, caused by external factors such as in the case of computer storage as proposed in [18]. This case of *dynamic* bin-packing adds further complexity to the algorithm, since extra care is potentially needed to handle bins which are suddenly empty or items causing fragmentation in the bin space.

In the case of First Fit, [18] shows that the complexity is increased, however marginally, due to the fact that the algorithm has to take care of active bins becoming empty.

V. Framework Architecture

In previous work [16], the *Intelligent Resource Manager* (IRM) system was designed as an extension of HIO based on the First-Fit bin-packing algorithm described above. Here, an overview of the bin-packing implementation and of the architecture of the extension components is presented.

A. Resource management with modified bin-packing

In order to maximize the resource utilization, the PE containers are scheduled with the help of a modified version of dynamic online bin-packing. Modelling PEs as bin items, workers as bins and the workload resource usage as the item size, the idea is that the algorithm can compute the optimal way to schedule the containers in order to keep the number of workers needed down while not congesting resources. Thus the IRM continuously runs the bin-packing algorithm on a queue of PEs waiting to be allocated. Considering the time- and space-complexity mentioned in IV-A and the one-dimensional scenario in this instance, as well as previous experiments on the HIO master [16, Figures 3,4], the impact of frequently running bin-packing is negligible.

Based on the outcome, HIO can determine where to host the containers and in addition whether more or fewer worker nodes are needed for the current workload autonomously. Through this feature, auto-scaling of worker nodes is achieved. To achieve auto-scaling of the PE containers the IRM monitors the streaming message queue to determine whether HIO is consuming stream requests at a high enough rate. If not, the IRM will queue more PEs in order to drive down the waiting time for stream requests. After a time of being idle, a PE will initiate a graceful exit-mechanism in order to free the resources and ensure that the PE will not receive any more requests.

As indicated in [16], the main metric used to represent resource utilization is the average CPU usage for a stream processing application, measured and updated as a sliding window. The average usage is directly used as the item size for the bin-packing algorithm. Furthermore, in order to not block the system when the workload pressure increases, a small buffer of idle workers are kept ready to accept stream requests.

Similarly to the bin-packing model in [18], some assumptions are in place in this implementation. Firstly, the case of "packing CPU utilization" does not suffer from fragmentation issues since this is just an abstraction of the slices of a period of time for which the CPU is allocated to a task. While too many processes will suffer from context switching, there is no need to reorganize the remaining space of the bin to fit new items.

Secondly, and more importantly, we assume that the actual CPU utilization does not fluctuate enough from the item size to cause any stress on the system while being packed to 100%. In our scenario, the desired state is that a worker, or bin, is using as close to 100% CPU as possible. In general a process may suffer from not having enough CPU, however the idea is that profiling the workload provides the optimal share of the

CPU for the process. Thus, a worker running at 100% CPU utilization is the desired behavior in this setting.

B. IRM architecture

In Figure 2, the architecture of the IRM extension is illustrated, showing the four main components of the system; the **container queue**, **container allocator**, **load predictor** and **worker profiler**. The following sections detail these components further.

There are a number of configurable parameters that control the behaviour of the IRM extension, which are briefly mentioned where relevant here. A table describing the parameters and showing the default value for each of them are available on the HASTE GitHub repository[8]. The choice of the default parameters were mainly based on heuristics from the original experiments and work in [16].

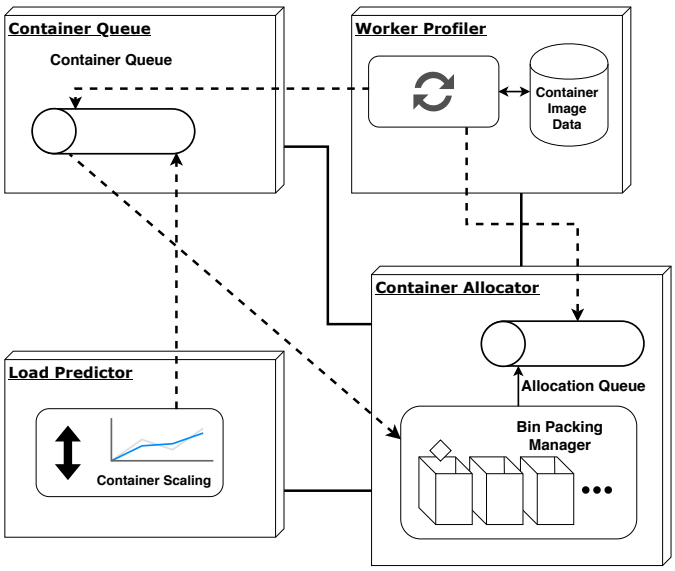


Fig. 2: Architecture overview of the IRM subcomponents. The dashed lines show flow of data and communication between components. Figure from [16, Fig. 2].

1) Container queue: Whenever a PE is to be created, it will first enter the *container queue*, a FIFO queue of container-hosting requests. Each request initially contains the Docker container image and tag, a time-to-live (TTL) counter, any metrics related to that image etc. The TTL counter is used in case the request is re-queued following a failed hosting attempt, in order to avoid infinitely repeating requests if they are not hosted on the first attempt.

While waiting in the queue, requests are periodically updated with new CPU usage metrics and finally processed by the periodic bin-packing algorithm. The queue holds requests both from auto-scaling decisions and manual hosting requests from users.

[8]https://github.com/HASTE-project/bin-packing-paper/blob/master/IRM_configuration_parameters.md

2) Container Allocator: The *container allocator* includes the bin-packing manager, responsible for performing the chosen bin-packing algorithm, and the allocation queue. As mentioned in Section V-A, in this model a worker VM represents a bin and the container hosting requests represent items. Active VMs indicate open bins, i.e. the VM is enabled as a host for PEs, with a capacity of 1.0 or 100% CPU (across all available CPU cores). The container requests have item sizes in the range $(0, 1]$, indicating the CPU usage of that PE from $0 - 100\%$.

The bin-packing manager executes the bin-packing algorithm, with the container queue as input, at a configurable rate based on this model, resulting in a mapping of where to host the queued PEs and how many worker WMs are needed to host these. The destination worker is attached to each container-hosting request, which is then forwarded to the allocation queue. As these requests are consumed the allocator attempts to start the PEs on the destination worker. In case a PE could not be started, for example if the target worker is a new VM still initializing, the request is reset to the initial state with the TTL reduced by 1, and it is sent to the container queue again.

3) Worker Profiler: To understand the resource utilization characteristics of the PEs, the *worker profiler* gathers runtime statistics from the workloads. The worker profiler is designed in two components; the first runs within the worker VMs, periodically measuring the current CPU usage for each running PE. The average CPU usage is calculated per Docker container and tag pair, and is then sent to the master VM. The second component in turn aggregates the information from all active workers as a new average across all VMs, wich is calculated as a moving average of the CPU utilization based on the last N measurements, N being arbitrarily configurable. Based on this information, the worker profiler provides an idea of the CPU utilization for PE container images that have been hosted on HIO previously. The average CPU is used by the bin-packing manager as the item size and the updated averages are propagated to container requests in the container and allocation queues.

4) Load predictor: The *load predictor* is responsible for tracking the pressure of the streaming requests to HIO. Looking at the length of the master node's backlog queue and its rate of change (ROC), the load predictor can determine if the rate of processing data streams is too slow and there is a need to add more PEs. The ROC provides predictions for the need to scale up, and scheduling PEs this way gives HIO time to set up additional workers and reduces the congestion of incoming stream messages.

The decision of scaling up is based on various thresholds of the message queue length and ROC. These thresholds are configurable, and there are four cases, resulting in either a large or small increase in PEs. In short, if the ROC or queue length are either above any of the related thresholds, this indicates that data streams are not processed fast enough. Reading the queue metrics is done periodically, and there is a timeout period after the decision to schedule more PEs before

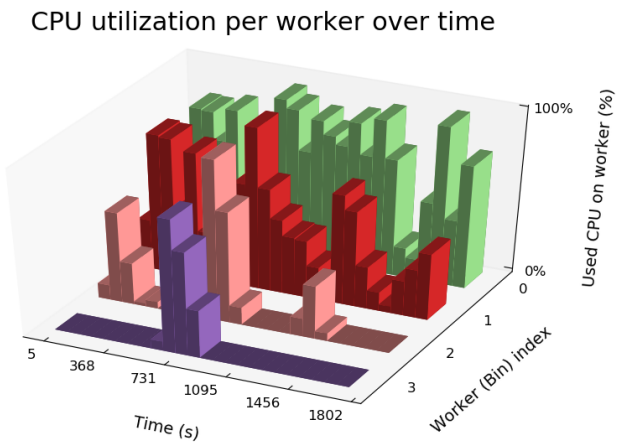


Fig. 3: The CPU usage from $0 - 100\%$ per worker over time in 3D. Image reused from [16, Fig. 8].

the load predictor can do this again.

VI. Results and Discussion

Experiments on the IRM-extended HIO system has shown promising results. As part of the thesis project [16], tests based on synthetic workloads were performed to evaluate the performance of the bin-packing implementation and the effect on resource utilization in HIO. The main outcome of these experiments are summarized in Section VI-A. Furthermore, new experiments have been conducted for a real-world image-analysis use-case in collaboration with AstraZeneca, where we benchmark and compare the resource utilization levels and strategies between HIO and Apache Spark Streaming. The results of these experiments are presented in Section VI-B.

A. IRM evaluation experiments on synthetic workloads

The IRM was tasked with profiling and scheduling workloads based on busying the CPU for specified usage levels and durations, mimicking a scenario where the bin-packing manager deals with items of various sizes and durations.

The main scenario that was experimented with included four different workloads all targeting 100% utilization of a single CPU core for various amounts of time. These were streamed in regular small batches of jobs and two peaks of large batches to introduce different levels of intensity in pressure to the IRM. Some of the results from these experiments are shown and briefly discussed here.

1) Efficient Resource Utilization: The results of the experiments indicate that the resource utilization may indeed benefit from the bin-packing approach. In Figure 3, the CPU usage per worker over time is shown in 3D-plots, giving an overview of the distribution of the jobs over the workers throughout the experiment. It is clear the workload is focused toward the lower index workers, leaving windows of time during which the higher index workers could be deactivated and the resources freed.

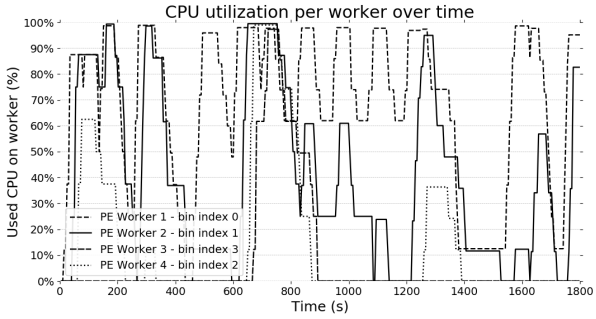


Fig. 4: Scheduled CPU usage per worker over time from the bin-packing manager. Plot data from [16, Fig. 9]. Synthetic workload dataset.

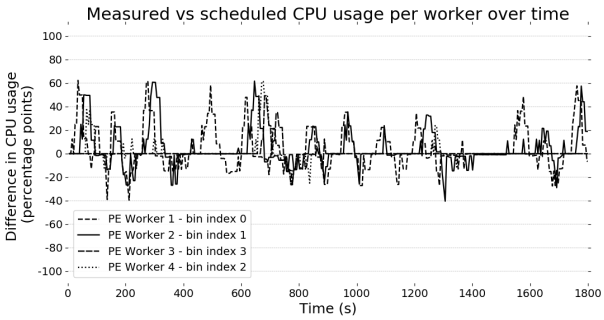


Fig. 5: Difference in percentage points between scheduled CPU usage and measured CPU usage over time per worker. Plot data from [16, Fig. 10]. Synthetic workload dataset.

Figure 4 shows the CPU utilization over time per worker as a 2D-plot, giving a better view of the utilization levels. In general, the utilization of the workers peak at between $90 - 100\%$ CPU usage. At this point, a worker can not fit further jobs and any following workloads are scheduled to a higher index worker, following the expected behaviour of the bin-packing algorithm.

2) Algorithm's Accuracy and Performance: Figure 5 shows a plot of the difference between the scheduled and measured CPU usage for each worker over time. The plot has a high amount of noise, and a likely hypothesis is that it is due to the delay in starting and stopping containers compared to when they are scheduled to start and stop. Another contributing factor is the irregularity in how often workloads are streamed, leading to PEs frequently starting and stopping.

Despite the erratic difference between the allotted and measured CPU usage plots, it is clear that the bin-packing algorithm does try to push the CPU utilization levels towards 100% on the workers. Moreover, the test scenario was designed to stream images at varying frequency, which impacts the ability of the framework to keep a constant high efficiency.

With this in mind, the results indicate overall that bin-packing provides an appealing approach to efficiently schedule containers in cloud computing contexts. Furthermore, the impact of the noisy error is hard to determine in a scenario with

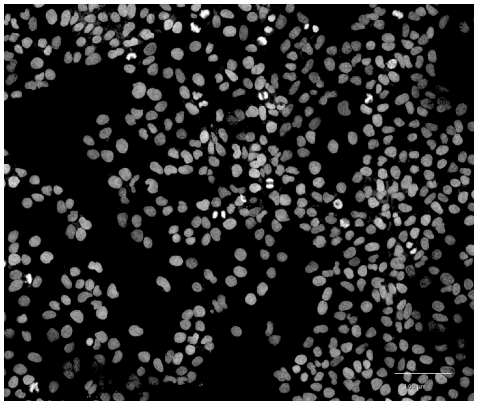


Fig. 6: Shown is a representative image of Huh7 liver carcinoma cells seeded at a moderate density, stained with the cell-permeable DNA binding dye Hoechst 33342, and imaged using a confocal microscope. The dye binds primarily to the DNA in the cell nucleus resulting in a fluorescence image that shows only the cell nuclei and not the cytoplasm or cell membranes. The scale bar shows $100\mu m$.

synthetic workloads and in heterogeneous distributed settings. More data from scenarios based on real use cases may help to better understand the actual benefits and drawbacks of the approach.

B. Image streams from quantitative microscopy

The data provided by AstraZeneca consists of a set of microscopy images (2 fields of view per well obtained using a Yokogawa CV7000 robotic confocal microscope fitted with a 20X Olympus air objective), as well as an analysis pipeline for CellProfiler[9]. Huh-7 cells (a hepatocellular lineage seeded at 6 different densities across a Perkin Elmer CellCarrier 384 well plate one day prior to imaging) were stained with nuclear dye prior to imaging (Hoechst 33342, ThermoFisher), and the CellProfiler analysis pipeline (Windows release 3.1.9) was created to count the number of nuclei and measure their areas. Due to variations in the images they take varying amounts of time to process, and the dataset includes a total of 767 images. Figure 6 shows an image from the dataset.

This CellProfiler pipeline was adapted to run in both HarmonicIO and Apache Spark Streaming environments; essentially the two versions both invoke an external process call to execute the pipeline. The next sections describe the experiments and results of running the image analysis in the two systems.

1) Apache Spark Streaming: For comparison with the system discussed, an Apache Spark Streaming application was developed and benchmarked for an equivalent image processing task. In terms of resource allocation, Spark will attempt to evenly distribute the load between available workers (to maximise IO performance and available memory) – this is in contrast to the approach adopted in HIO with the IRM extension which will attempt to saturate the resources of each node in turn as the workload increases. This difference in strategies is clearly visible in Figures 4 and 7.

With an adapted *Spark File Streaming* source, for each new image, CellProfiler is invoked as an external process to analyze the image, using the pre-configured image processing pipeline from AstraZeneca.

The image file names need to be passed to CellProfiler, but they are not easily available from the Spark APIs via the HDFS driver. Additionally, loading the image file contents into RDDs introduced an unnecessary performance penalty, so we modified Spark's `FileInputDStream` class so that the DStream contained only the file paths, which are ultimately passed to CellProfiler for execution (CellProfiler then reads the image data via the NFS share).

Spark is often used in conjunction with 3rd party resource managers such as Mesos (http://mesos.apache.org/), and YARN (http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html, Hadoop's resource manager). These frameworks manage a shared pool of resources among applications such as Hadoop, Spark, Flink, Storm, etc. Instead, we used a so-called *standalone* Spark deployment, where all resources are allocated to Spark, which handles the allocation among running Spark applications.

Some experimentation was needed to achieve satisfactory auto-scaling behaviour. There is support for *dynamic allocation* (that is, scaling the Spark application within the cluster) *specifically for streaming*, since Spark 2.0.0, (configured with the settings `spark.streaming.dynamicAllocation.*`), taking into consideration the batch processing and addressing other issues (https://issues.apache.org/jira/browse/SPARK-12133). However, our initial attempts to achieve satisfactory auto-scaling with this approach were problematic, because it begins to scale up only when a batch is completed. So, when the system is initially idle (with a single executor), the initial set of images for a 5 second batch interval (approximately 50 images), each with having an execution time of 10-20 seconds, meant that the first batch takes minutes to execute, leaving the other available cores in the cluster unused.

Because the images are processed by invoking CellProfiler as an external process, it is the minimum unit of parallelism. For this reason, we resorted to using the older *dynamic allocation*, which confusingly has a similar name to the new streaming-specific functionality, with an `spark.dynamicAllocation.executorIdleTimeout` of 20 seconds. Under this auto-scaling mechanism, a request for additional executors is triggered by a backlog of Spark tasks; with idle executors removed after a timeout. In practice, triggering on a task backlog (rather than a complete batch), results in a lower latency when scaling up, and made a big difference for our workload – because we don't need to wait for completion of the batch. With configuration of various timeouts (and limits on min/max executor counts), this approach is a robust, application agnostic approach to scaling, which works well for wide range of workloads. It requires

[9]www.cellprofiler.org

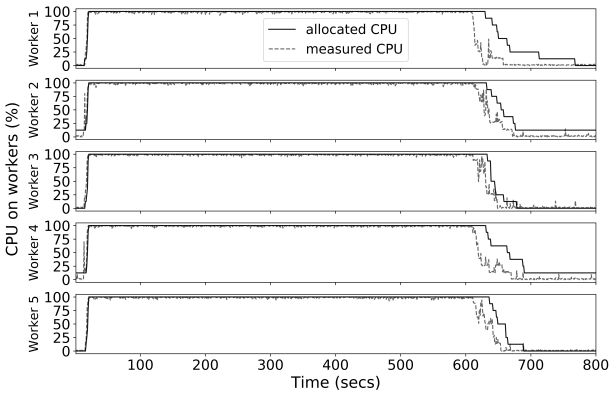
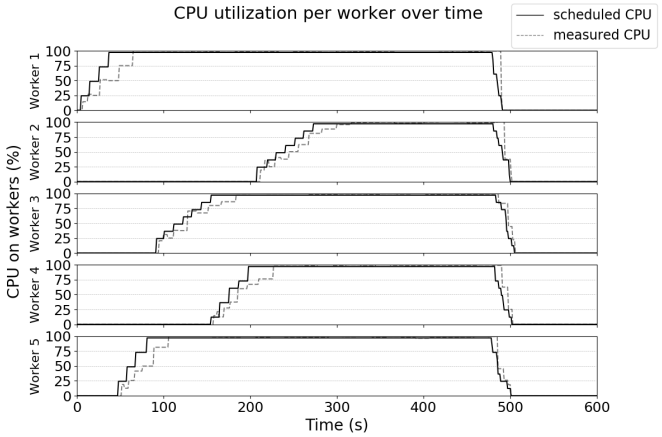
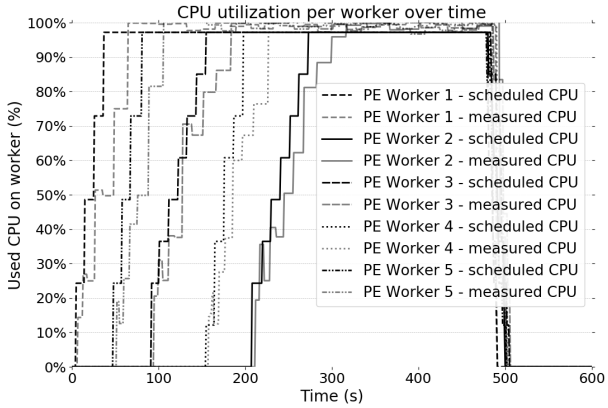


Fig. 7: Spark allocated executor cores vs. actual CPU usage. Huh7 liver cells dataset.



(a) CPU usage per worker over time, stacked plots.



(b) CPU usage per worker over time, common plot.

Fig. 8: Scheduled vs measured CPU utilization per HIO worker over time, plotted in two different ways. The worker names are not related to the bin indexes. Huh7 liver cells dataset.

application experts to roughly know resources required for each job which provides basis for the auto-scaling strategy.

This begins scaling during the first batch. We also raised the `spark.streaming.concurrentJobs` setting from 1 to 3, so that other cores could begin processing the next batch while waiting for the 'tail' of images within the job to each finish their 10-20 seconds of processing. This gave satisfactory auto-scaling performance.

The application was written in Scala, and is available online (including all the scaling settings)[10]. The source directory was mounted as an NFS share on all the machines in the Spark cluster, because the image sizes (order MB) are too small to warrant the overhead of HDFS. CellProfiler and its dependencies were installed on all the machines in the Spark cluster.

Spark Version 2.3.0 was used. The cluster was deployed to SNIC science cloud [23], a computing service for Swedish academia. The cluster consisted of 1xSSC.xlarge (for the spark master), 5xSSC.xlarge (for the workers), and 1xSSC.small for the virtual machine hosting the images. For the benchmarking, the elapsed system CPU usage on all the workers was polled with `top` (we take the sum of user and kernel CPU time), and the number of executor cores was polled via the Spark REST API. The clocks on all machines were initially synchronised with `ntpdate`. By combining the log files, Figure 7 was generated showing the real CPU used (shown number of cores) and the total cluster executor cores reported by the REST API.

A number of phenomenon are visible in Figure 7. We clearly see the executor cores scale up and down for the batch processing. The system scales to use all the available 40 worker cores in the cluster. The trace begins before image streaming begins.

2) HarmonicIO with the IRM extension: For testing the CellProfiler analysis pipeline in in the HIO framework, the Stream Connector was used to stream the entire collection of images as a single batch of one image per message. The PEs were designed to initiate the pipeline with this single image as input.

The machine setup was identical to the setup for the Spark experiment. The VMs for HIO were deployed on the SNIC science cloud, with one master node (SSC.xlarge), five worker nodes (SSC.xlarge) and one client (SSC.small). The IRM configuration for HIO uses the uses the default values for all parameters, as noted in Section V-B, with the additional worker parameters `report_interval` and `container_idle_timeout` both at 1 second.

The experiment shown in the following figures was performed after the HIO cluster had seen the workload previously and thus had an initial guess of the workload CPU usage.

Starting with the CPU utilization, Figure 8 illustrates how the bin-packing manager scheduled the CPU usage across the workers throughout the run. As visible in both plots, the workers are scheduled to use nearly 100% of the CPU before the auto-scaling drives workloads to the next worker. Note that the worker numbers are not related to the underlying bin index.

[10] github.com/HASTE-project/bin-packing-paper/blob/master/spark/spark-scala-cellprofiler/src/main/scala/CellProfilerStreaming.scala

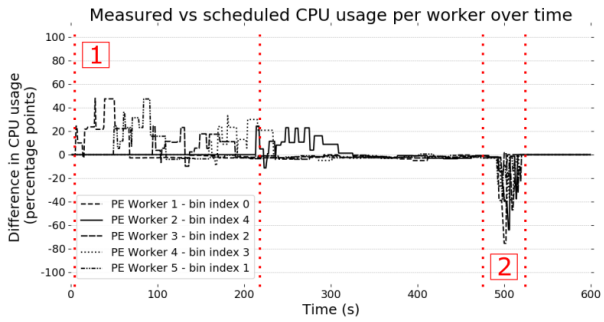


Fig. 9: Difference in percentage points between scheduled and measured CPU usage in percentage points. Huh7 liver cells dataset.

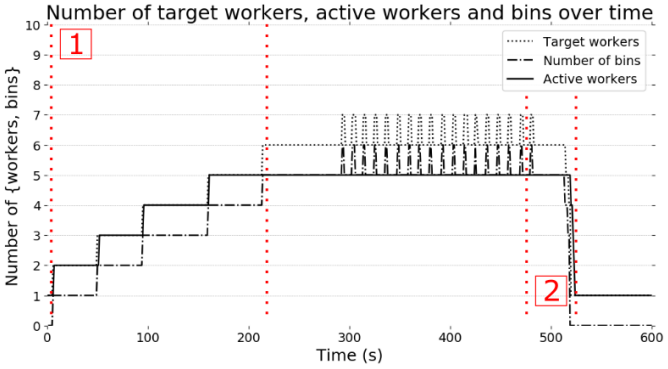


Fig. 10: Number of currently active workers and target number of bin and active workers over time. Huh7 liver cells dataset.

Also visible in both plots is a delay in measured CPU usage from when the bin-packing manager schedules it. The next plot in Figure 9 shows the difference between the scheduled CPU and the measured CPU usage. Finally Figure 10 shows the number of currently active workers and target number of bins and active workers in the HIO cluster throughout the experiment.

If we compare the plots in Figures 8(b), 9 and 10, we can see that the bumps in difference between measured and scheduled CPU for each worker coincide with the periods during which the bin-packing manager increases the number of PEs on that worker, further highlighted by the red lines in the latter plots. Building on the hypothesis from the analogous plots from the synthetic workload tests, this confirms our initial theory; since the PEs will take a few moments to start processing incoming data after having been scheduled, there will be a difference between scheduled CPU and measured CPU usage (phase 1, highlighted with the first two dotted red lines in Figures 9 and 10).

After this period the error settles close to 0, indicating that the scheduled workloads are quite accurately matching the actual workload and the CPU is close to ideal from a resource utilization-maximizing perspective. The sudden large decrease shows the inverse, where the containers start shutting down from being idle in rapid succession (phase 2, highlighted with the last two dotted red lines in Figures 9 and 10).

Thus, we argue that despite the lag of the measured CPU usage reaching the scheduled utilization due to startup and shutdown time of containers, the bin-packing algorithm is evidently efficient in the scheduling of CPU resources. Basing the IRM on bin-packing we manage to schedule the resources close to 100% utilization levels, while the applications running on the workers are provided the share of CPU that they require.

Worth to note is that as illustrated in Figure 10, the IRM would have scheduled more workers to handle the workload if they were available. In order to make an accurate comparison of the efficient utilization of available resources with the Spark Streaming framework, we have restricted both of the frameworks to 5 workers. The periodic attempts to increase further are due to the IRM attempting to scale up, scheduling more PEs than can fit on the available 5 workers. These attempts will fail while the IRM constantly tries again to scale up until the queued images are processed.

An interesting observation can be seen in comparing Figure 7 and Figure 8. The plots show that the time taken to complete the entire workload was less in the HIO setup for this scenario. At the same time, we can also see that HIO was slower in scaling up all the workers, only spreading to the next worker after reaching full CPU capacity on the previous. In contrast, Spark quickly scaled up to utilize the full CPU capacity on all workers at the same time. This highlights a benefit that can be achieved by adopting a more resource-efficient scaling strategy under the right conditions; in this scenario, the workers that are not used in the HIO system could potentially have been utilized for other tasks or inactive, saving on computing resources, energy consumption and cost.

Another important observation is that judging by the plots from Section VI-B2, the performance of the IRM has been much better during the experiments with the real usecase compared to the experiments with synthetic workloads discussed in Section VI-A. The main difference between the two cases was that all the images were streamed in a large batch as opposed to periodic small batches. Thus HIO was allowed to really push the CPU usage to it's intended level with constant processing.

VII. Conclusion and Future Directions

The efficient management of resource utilization in streaming frameworks is a non-trivial task. The current streaming frameworks already have multiple parameters to tune to provide reliable processing of large datasets, and efficient resource utilization further adds to the complexity of the frameworks. It is nonetheless an important problem and must be addressed. More effort is needed both from industry and academia to explore methods of resource utilization optimization that are simple, effective and capable of handling unexpected scenarios with minimal underlying assumptions.

The presented approach for efficient resource utilization based on online bin-packing fulfills these requirements in

the setting of large-object data streaming analysis. Our results illustrate efficient scheduling of computing resources based on two very different use cases, where in both cases the framework requires no a priori information about the workloads from the users. The presented experiment measurements highlight that despite the CPU utilization slightly lagging behind the CPU scheduling, HarmonicIO with the IRM extension can handle this and offers a stable and efficient processing framework. The potential is especially visible in the real use-case experiment, during which the images are streamed at a high frequency allowing the resources to be used consecutively.

As mentioned earlier, HarmonicIO is designed to address the needs for processing large datasets based on relatively large individual objects. It is a specialized streaming framework that is well suited for scientific workflows. The presented solution based on dynamic online bin-packing fits well with HarmonicIO and improves the framework by adding resource efficient autoscaling, making it comparable to frameworks used in the industry such as Apache Spark Streaming.

In the future, we would like to further extend our approach with multi-dimensional bin-packing. The motivation for this is to be able to profile and schedule workloads based on other resources than only CPU, such as RAM, network usage, or even variations of CPU metrics like average, maximum etc. This would allow us to handle more challenging use cases other than the image analysis workflows covered so far.

Acknowledgements

This research was undertaken as part of the HASTE project. The HASTE Project (http://haste.research.it.uu.se/) is funded by the Swedish Foundation for Strategic Research (SSF) under award no. BD15-0008, and the eSSENCE strategic collaboration for eScience.

The authors would like to acknowledge and thank the SNIC Science Cloud (Project No. SNIC 2020/20-42) for providing the cloud resources used to host the experiment environments.

References

[1] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski, and M. Woniak, "Ensemble learning for data stream analysis: A survey," *Information Fusion*, vol. 37, pp. 132 – 156, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1566253516302329

[2] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, P. P. Jayaraman, J. Hosking, and E. Abbasnejad, "Resource usage estimation of data stream processing workloads in datacenter clouds," 2015.

[3] V. M. A. Souza, D. F. Silva, J. Gama, and G. E. A. P. A. Batista, *Data Stream Classification Guided by Clustering on Nonstationary Environments and Extreme Verification Latency*, pp. 873–881. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611974010.98

[4] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Autoscaling in data stream processing applications: A model-based reinforcement learning approach," in *New Frontiers in Quantitative Methods in Informatics*, S. Balsamo, A. Marin, and E. Vicario, Eds. Cham: Springer International Publishing, 2018, pp. 97–110.

[5] B. Blamey, A. Hellander, and S. Toor, "Apache spark streaming and harmonicio: A performance and architecture comparison," *arXiv preprint arXiv:1807.07724*, 2018.

[6] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[7] P. Torruangwatthana, H. Wieslander, B. Blamey, A. Hellander, and S. Toor, "Harmonicio: Scalable data stream processing for scientific datasets," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 879–882.

[8] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 104 – 111, 2012.

[9] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 117–126. [Online]. Available: http://doi.acm.org/10.1145/1555228.1555261

[10] L. Tomás and J. Tordsson, "An autonomic approach to risk-aware data center overbooking," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 292–305, 2014.

[11] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2647–2660, 2014.

[12] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, Sep 2007. [Online]. Available: https://doi.org/10.1007/s10586-007-0035-6

[13] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 705–718, Feb. 2018. [Online]. Available: https://doi.org/10.14778/3199517.3199521

[14] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171 – 185, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17326821

[15] "Haste: Hierarchical analysis of spatial and temporal data," http://haste.research.it.uu.se/, (Date last accessed 9-August-2020).

[16] O. Stein, "Intelligent Resource Management for Large-scale Data Stream Processing," Master's thesis, Uppsala University, Sweden, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-391927

[17] S. S. Seiden, "On the online bin packing problem," *J. ACM*, vol. 49, no. 5, pp. 640–671, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/585265.585269

[18] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, "Dynamic bin packing," *SIAM Journal on Computing*, vol. 12, no. 2, pp. 227–258, 1983.

[19] L. Epstein, L. M. Favrholdt, and J. S. Kohrt, "Comparing online algorithms for bin packing problems," *Journal of Scheduling*, vol. 15, no. 1, pp. 13–21, 2012.

[20] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[21] C. C. Lee and D. T. Lee, "A simple on-line bin-packing algorithm," *J. ACM*, vol. 32, no. 3, pp. 562–572, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.3833

[22] G. Gambosi, A. Postiglione, and M. Talamo, "Algorithms for the relaxed online bin-packing model," *SIAM journal on computing*, vol. 30, no. 5, pp. 1532–1551, 2000.

[23] "Snic science cloud," https://cloud.snic.se, 2017, (Date last accessed 19-November-2019).