RESOURCE CONFIGURATION DECISIONS FOR BULK SYNCHRONOUS PARALLEL JOBS

By

Evan Lei Wang

Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
in

Computer Science

August 13th, 2021

Nashville, Tennessee

ACKNOWLEDGMENTS

Thank you to Yogesh Barve, for providing helpful resources and answering my questions at every step.

Thank you to my advisor, Aniruddha Gokhale, for helping guide my research and pointing me in the right direction.

Thank you Anirban Bhattacharjee, for speaking to me about your research and giving advice about my problem space.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

Introduction

In the Bulk Synchronous Parallel (BSP) model, a job consists of multiple tasks, which run over a series of supersteps [1]. The tasks perform computations concurrently and need to synchronize with each other at the end of each "superstep", as illustrated in Figure 1.1. This allows individual tasks to communicate with each other and share the results of their previous computations.

Since tasks need to wait for all other tasks to complete, the overall duration of the computation depends on the slowest task at each step. We would like to prioritize our resources towards these stragglers (by giving them more CPU cores or memory) in order to speed up their execution and reduce the wait-time of other tasks.

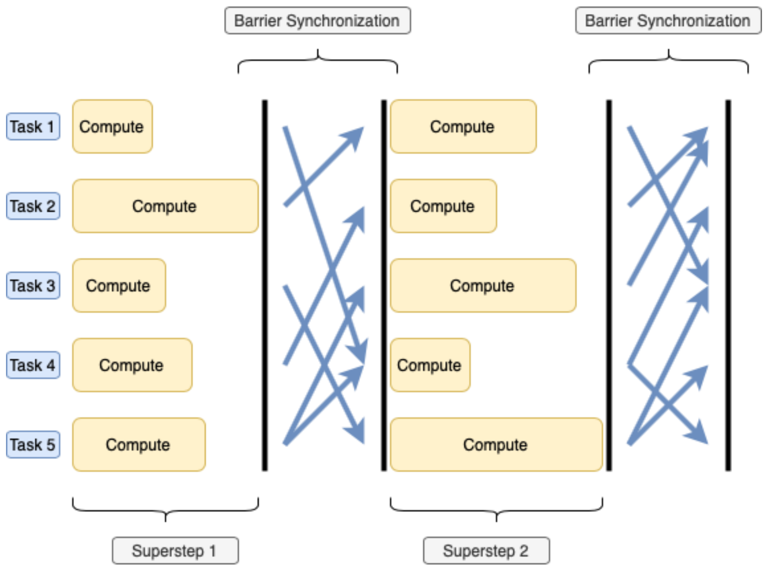


Figure 1.1: Bulk Synchronous Parallel job.

When we schedule a Bulk Synchronous Parallel job, we must define a resource configuration (a partition of available resources among all the tasks). This resource configuration is used to gang-schedule tasks onto nodes (usually virtual machines) that have sufficient resources. We hope to optimize our available resources by creating resource configurations that minimize the run time of the slowest job at each time step (and therefore the run time of the overall job).

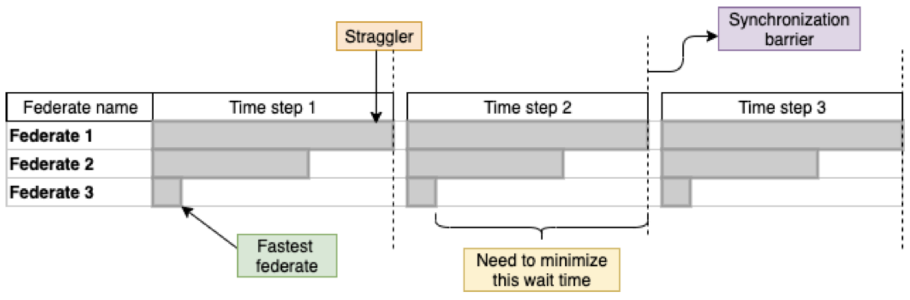
Table 1.1 provides an example of a scheduled Bulk Synchronous Parallel job, with its resource configuration and node mapping. Note that in the this current research, our resource configurations only allocate CPU cores. Adding memory considerations to this research is listed as future work in Section 7.1.

Task	CPU Cores	Memory	Scheduled Node
Task 1	6 Cores	11GB	Virtual Machine 1
Task 2	12 Cores	9GB	Virtual Machine 3
Task 3	3 Cores	1GB	Virtual Machine 1
Task 4	9 Cores	3GB	Virtual Machine 2
Task 5	7 Cores	5GB	Virtual Machine 4

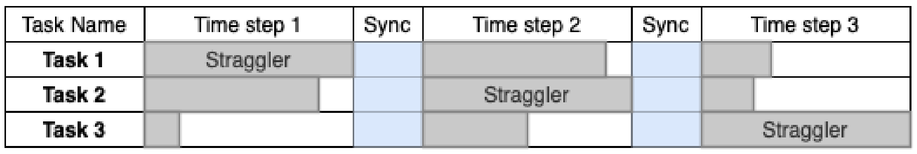
Table 1.1: Example of scheduled Bulk Synchronous Parallel job with resource configuration.

The goal of this thesis is to make resource configuration decisions that reduce the overall computation time of Bulk Synchronous Parallel jobs. This research extends a recent paper which introduces an algorithm for computing optimal resource configurations for co-simulations [2]. However, [2] simplifies resource configuration decisions by assuming that task workloads are static across time steps. This means that each task performs the same computation at every time step, so their execution time remains constant. The authors compute a resource configuration that optimizes execution time for a single set of task workloads. Since these workloads are static, this single configuration can be reused for the entirety of the job.

This thesis removes the assumption that task computations are static, adding an additional layer of complexity to the problem. If task workloads are dynamic, then a task may receive a small computation at an earlier time step and a much larger computation at a later time step. Therefore, the resource configuration will need to adapt to these changes.



(a) Static workload example from [2]



(b) Dynamic workload example

Figure 1.2: Example of job performance using static workloads (a) vs. dynamic workloads (b) for each task.

The assumption made in [2] is illustrated by Figure 1.2. In the static co-simulation example, task (federate) 1 is the slowest task at each time step. However, in the dynamic example, we see that the workload of each task can change. Task 1 starts out as the slowest task (straggler) at time step 1, but Task 2 becomes the slowest task at time step 2, and Task 3 is the slowest at time step 3. The dynamic example may be more indicative of the real world, where workloads can follow trends or patterns.

This thesis addresses the challenge of choosing resource configurations over multiple time steps and adapting these configurations to task workload fluctuations. The solution in this paper introduces a greedy algorithm for computing high-performing resource configurations across multiple time steps (with dynamic workloads). It leverages this algorithm to create resource configuration plans that balance performance improvement with reconfiguration costs. This paper presents multiple different strategies for making resource configuration plans (including dynamic programming and reinforcement learning approaches) and evaluates them in a simulated environment.

The thesis is organized as follows:

- Chapter 2 highlights the problem and lays out the requirements for this research.
- Chapter 3 discusses related work in this area and their applications to this research.
- Chapter 4 describes the overall design for the research and summarizes the individual components. It also introduces Algorithm 1, which computes resource configurations for dynamic workloads.
- Chapter 5 gives an in-depth description of the resource configuration planning strategies used in this research.
- Chapter 6 describes the experimental setup and simulation results for each of the resource configuration planning strategies.
- Chapter 7 concludes the thesis and describes possible avenues for future work.

CHAPTER 2

Problem Statement

2.1 Use Cases

The Bulk Synchronous Parallel model can be used for computations where work may be done in parallel, but computation results need to be synchronized periodically. An important real-world implementation of the Bulk Synchronous Parallel model is Google Pregel [3]. Pregel is a graph processing framework where computations are done concurrently at individual vertices. After their computations, vertices are able to send messages to each other which can be accessed at the beginning of the next superstep. The wait time for vertices to send their messages is the barrier synchronization aspect of the Bulk Synchronous Parallel model.

Another very interesting use case for the Bulk Synchronous Parallel model is predictive digital twins, or, more specifically, co-simulations. A predictive digital twin is a digital copy of a physical system. By running simulations on the digital twin, we can perform system analysis and make predictions about the behavior of its physical counterpart. For example, we might have a digital twin of an aircraft [4]. We would then be able to simulate different flight paths and get information about how much damage the aircraft will sustain. However, Willcox et al. explain that accurate simulations for systems as complicated as an aircraft would require a massive number of variables. As a result, they generally take a divide and conquer approach which involves the splitting up the system into much smaller components (such as a piece of the wing). Each individual component has a co-simulator, which simulates the state of that component. These co-simulators perform their own computations, but they will need to synchronize with other co-simulators at the end of the time step to get updated values for the state of the overall aircraft. After this barrier synchronization, the co-simulators can continue to preform calculations for the next time step of the simulation.

Co-simulations, like the ones used in this predictive digital twin example, follow the Bulk Synchronous Parallel model where individual tasks execute their own operations and synchronize before the next superstep.

2.2 Problem Specification

In this thesis, we aim to reduce the overall execution time of Bulk Synchronous Parallel jobs. We assume that we are provided a fixed number of resources for an individual job which are scattered across multiple computing nodes (or virtual machines). Each task runs as a Docker container on a node that has enough resources to accommodate it [5]. Because the duration of a job at each time step depends on the slowest task at that time step, we want to provision our resources in a way that prioritizes larger tasks. This will improve the performance of the job by speeding up stragglers and reducing the wait time of other tasks.

2.2.1 Task Workload Variation

Unlike previous work, we assume that tasks can have dynamic workloads. This means that tasks may have computations (or workloads) of different sizes at different time steps. In this research, we model the fluctuations of task workload sizes using time-series datasets. These datasets can provide realistic examples of real-world trends and patterns.

We define workload size as a description of the number of operations that a task needs to perform. A larger workload size means more operations and a longer computation time for the same task. As a result, we would like to provide more resources to a task when it has to perform more computations and fewer resources when it has to perform less computations.

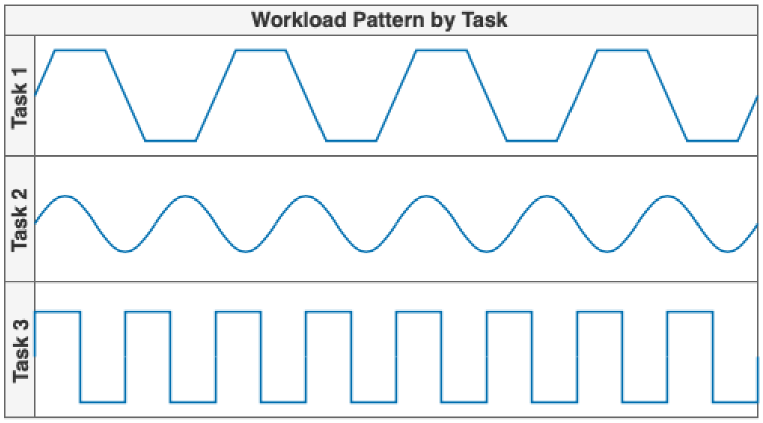


Figure 2.1: Example of workload fluctuation patterns for each task.

2.2.2 Reconfiguration Costs

Lastly, we assume that there is some cost associated with changing resource configurations. In order to change resources provided for a task, we would have to alter the resources for the container. If multiple containers on a node all require more CPU cores, the node may no longer have enough cores to satisfy the requirements and some containers would need to be moved. This would require tearing down the containers and scheduling them onto a new node. There is overhead associated with recreating these containers. Additionally, in the co-simulation use-case, we have to preserve the internal state of the task (or co-simulator in this case). This requires check-pointing the container, which also results in its own overhead. If no such cost existed, it would be optimal to create new resource configurations as frequently as possible. Each resource configuration would be created specifically for the workload sizes of that time step.

For this thesis, we focus on the resource configuration aspect, and we allow the Kubernetes scheduler to handle the scheduling of containers onto specific nodes [6]. We assume that if we need to adjust the resource configuration, the pod will automatically be checkpointed. When we decide to change the resource configuration, we wait for tasks to synchronize and create a new configuration from scratch. Then, we pause the job, checkpoint all the containers concurrently, and recreate them with the new configuration. In this thesis, we represent this cost with a single, constant checkpoint cost.

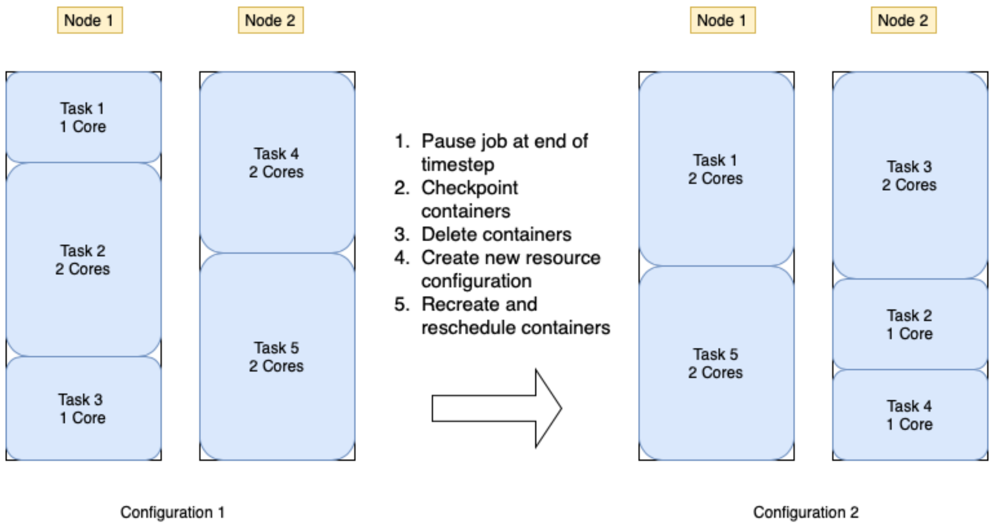


Figure 2.2: Example of changing configurations from Configuration 1 to Configuration 2.

2.3 Problem Requirements

For this problem, a solution should minimize the overall computation time of the Bulk Synchronous Parallel job (including reconfiguration costs). The solution should be able to:

- Learn from historical workload trends.
- Create resource configurations that minimize runtime over multiple time steps.
- Create a configuration plan (plan for how frequently we reconfigure resources) that balances task performance and reconfiguration costs.

CHAPTER 3

Related Work

3.1 Resource Configurations for Co-Simulations

This thesis extends research from a previous paper, [2], which defines an algorithm for creating resource configurations for co-simulations. In this paper, Barve et al. first perform workload profiling to determine the computation time of individual co-simulators with different numbers of CPU cores. Then, starting from a default configuration, the authors increment resources for the slowest running task until all service level objectives (SLOs) are met at minimal cost. This results in a resource configuration which optimizes co-simulation performance.

However, this paper makes the assumption that individual tasks always have static workloads. If we relax this assumption, like in Figure 1.2, then the algorithm will no longer find an optimal resource configuration across multiple time steps.

3.2 Time-Series Forecasting

In [7], Bhattacharjee et al. perform auto-scaling for deep-learning models based on the number of client requests. This request count exhibits time-series variation, so the authors must adapt to fluctuations in workload.

The authors also use profiling to gather data on the deep learning model's performance with different resource configurations. Then, using time-series forecasting models, they make predictions for the number of requests they will receive at future time steps. Based on these predictions and the profiling data, Bhattacharjee et al. decide how to vertically and horizontally scale resources for the deep learning model.

3.3 Resource Configuration Planning

[8] presents a strategy for making reconfiguration plans that minimize a given cost. The cost is defined as a combination of response time above SLO, cost of changing the number of machines, and the costs of the machines themselves. The cost of changing the number of machines is similar to the checkpoint or reconfiguration cost of our own problem space.

Like the previous paper, Roy et al. leverage time-series methods to make predictions about future workloads. Based on these predictions, the authors make decisions about scaling resources. At each time step, they define the possible actions as increasing the number of machines, decreasing the number of machines, or maintaining the current number. Then, using a receding horizon approach, the authors find a configuration plan that minimizes the cost over a predetermined window [9]. The configuration plan is defined as a sequence of actions (one at each time step) over the entire window. Using this plan, they are able to decide what changes to need to be made to the resource configuration at every time step. Since there are a finite number of actions and time steps, Roy et al. are able to use tree search to search through all possible sequences of actions to find one that minimizes the cost.

3.4 Applications to the Bulk Synchronous Parallel Problem Space

This thesis applies many of the ideas from the previous papers to the Bulk Synchronous Parallel model:

- It builds on the resource configuration algorithm from [2] in order to create resource configurations that improve straggler performance over multiple time steps.
- It uses workload profiling to model task performance under various conditions.
- It applies the pattern of combining profiling data with time-series predictions from [7] and [8] to predict execution times for each task (Figure 4.3).
- It uses the receding horizon approach (also known as model predictive control) to create a resource configuration plan over a predefined horizon.

3.5 Key Differences for the Bulk Synchronous Parallel Problem Space

As stated in Chapter 2 and Section 3.1, this research removes the assumption made in [2] that task workloads are static. This added complexity means that we need to adapt our resource configurations to changing workload conditions.

Both [7] and [8] provide solutions for scaling resources proactively against changing workloads. However, unlike the problem spaces in these papers, Bulk Synchronous Parallel jobs consist of multiple, separate tasks. Optimizing the runtime of a BSP job means that we need to ensure all of our tasks are running as quickly as possible. When we adjust our resource configurations we need to consider the performance of each task. Different tasks may have different performance models, so we need to conduct extensive workload profiling for each task. Additionally, tasks can follow their own unique time-series fluctuations, so workload forecasts need to be made independently for each task.

Another consideration is that all tasks need to synchronize at the end of the time step, so our performance depends on the execution time of the slowest task. Instead of only scaling resources for a single task, this problem space requires dividing a constant amount of resources among multiple tasks. Provisioning more resources for a task means that other tasks will have less resources available. The strategy used in [8] presents a method to search through all combinations of resource configurations over a set time frame. However, our resource configurations require us to provision resources separately to each task, instead of simply scaling up and scaling down the number of machines. There are an exponential number of ways for us to divide available resources between tasks, making it infeasible to test all possibilities. As a result, this research needs to reduce the number of configurations and actions that are considered.

CHAPTER 4

Research Design

To address the requirements discussed in Section 2.3, this research presents the following resource management model for Bulk Synchronous Parallel jobs. The model consists of these four major components as illustrated in Fig 4.1:

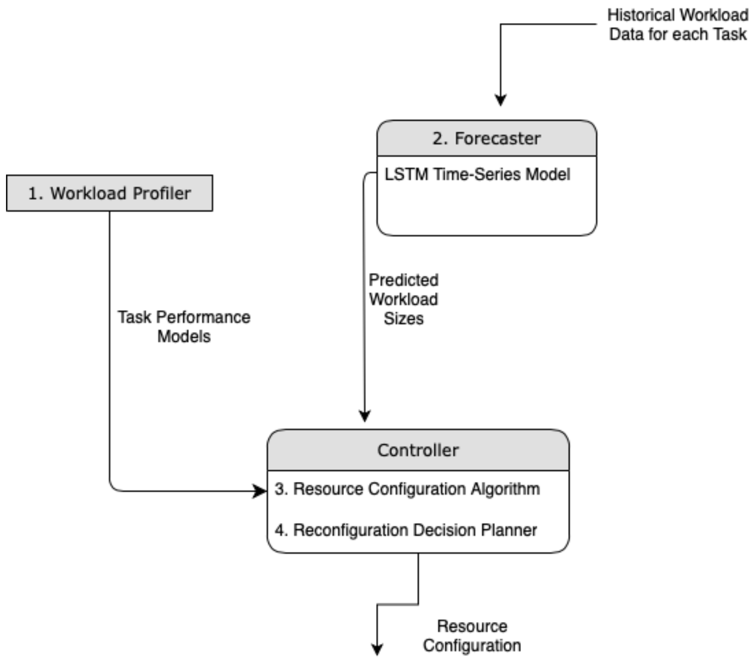


Figure 4.1: Outline of the research design.

1. A workload profiler that builds performance models for each BSP task. These models can predict the execution time of that specific task.

2. Time-series models for each task to predict future workloads.

3. An algorithm for creating a resource configuration for the next N time steps.

4. A planner that decides how frequently reconfiguration happens and how long a configuration is kept.

These components are combined to create a system that dynamically manages resource configurations for Bulk Synchronous Parallel jobs.

4.1 Workload Profiling

For the workload profiling, we run each task with a variety of resource configurations and workload sizes. This provides us with information about how tasks perform with different combinations of inputs. Tasks may respond to changes differently; for example, some tasks can benefit more than others from increasing CPU cores.

4.1.1 Benchmark Tasks

The eight tasks in our Bulk Synchronous Parallel job are stress-ng benchmarks, which execute a wide variety of operations for stress testing the machine [10].

Task Name	Description
affinity	Rapidly change CPU Affinity
atomic	Exercise GCC __atomic_*() built in operations
bsearch	Performs binary search on sorted array
cap	Make calls to capget(2) system calls
chmod	Change file mode bits of a single file with chmod(2) and fchmod(2) system calls
memcpy	Copy data from shared region to a buffer
vecmath	Perform calculations on 128 bit vectors
zero	Make reads on /dev/zero

Table 4.1: The stress-ng benchmarks that are used as tasks in our Bulk Synchronous Parallel job.

4.1.2 Profiling Results

Each task is run with different CPU configurations, from 1000M to 9000M CPU shares (with 500M increments). CPU shares are Kubernetes way of provisioning CPU cores to containers [11]. 1000M CPU shares is equivalent to 1vCPU.

We define workload size as an integer between 10 and 50. It is a multiplier on the number of operations the task must execute; therefore, a workload size of 50 means that a task needs to compute 5 times as many operations as a workload size of 10. For each CPU configuration, the task is run twice with workload sizes ranging from 10 to 50 (with increment of 3). The tasks are run as Kubernetes jobs, and the average of the execution times is recorded.

The heat maps in Figure 4.2 show the profiling results for each of our tasks. As we would expect, larger workload sizes lead to higher durations, while more CPU shares lead to lower durations.

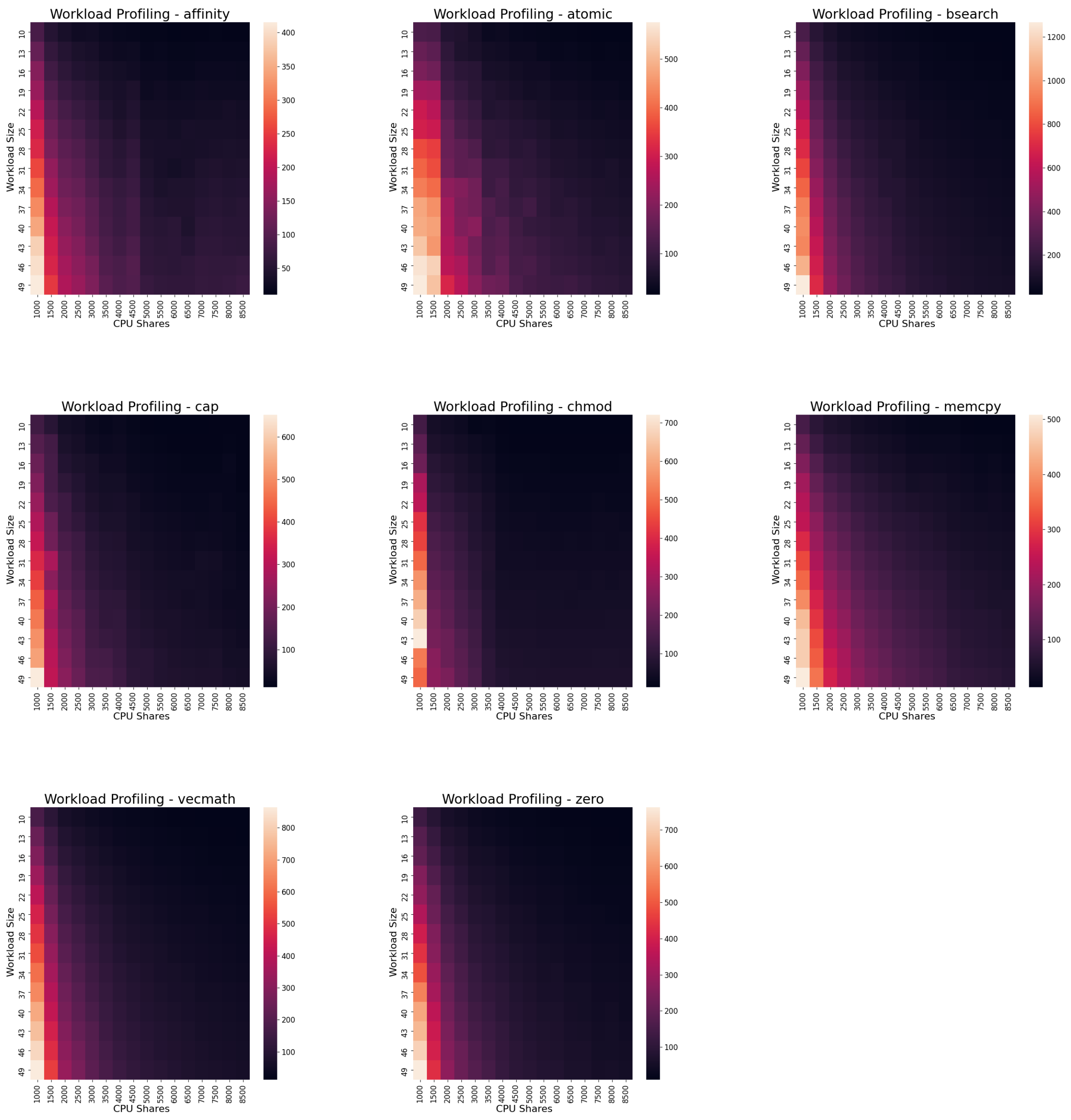


Figure 4.2: Workload profiling heatmaps for each of our benchmarks.

4.1.3 Task Performance Models

By analyzing the profiling results, we can learn how a task responds to changes in CPU resources and workload size. This research uses these results to create a model for the task's execution time. These models can predict task performance given any combination of workload size and CPU resources.

We operate on the basic assumption that a task's execution time will be close to previous iterations with similar workload sizes and CPU resources. Those iterations will correspond to points that are nearby on the grid of (workload size, CPU resource) data-points. As a result, the model looks for nearby points on our grid of profiling results and uses them to construct its predictions. For the actual model prediction logic, this thesis uses interpolation libraries from scipy [12].

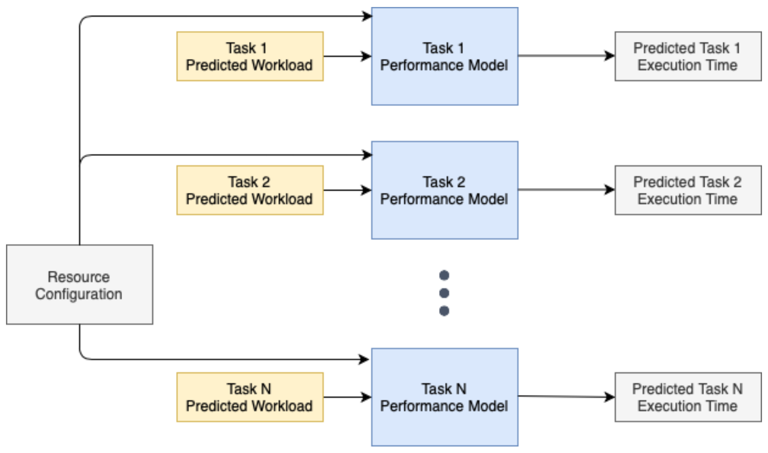

Task 1 Predicted Workload
Task 1 Performance Model
Predicted Task 1 Execution Time
Task 2 Predicted Workload
Task 2 Performance Model
Predicted Task 2 Execution Time
Resource Configuration
Task N Predicted Workload
Task N Performance Model
Predicted Task N Execution Time

Figure 4.3: The task performance model for each of the N tasks. Each model predicts task execution time given a resource configuration and workload.

4.2 Time-Series Modeling

For each task, we use time-series datasets from the Numenta Anomaly Benchmark to model fluctuations in workload sizes [13]. These datasets provide real-world time-series variation that exhibit realistic trends and patterns. Each of our eight tasks has a separate sequence of time-series data for simulating changes in their workload. Every task also has its own time-series model that trains on historical patterns from that task's dataset.

4.2.1 Datasets

There are three main datasets used for the time-series modeling:

- **NYC Taxi** - Number of passengers for NYC taxi cabs. Data recorded every 30 minutes.
- **Ambient System Temperature Failure** - The temperature in an office setting. Data recorded hourly.

- **Artificial Daily Small Noise** - Artificially generated dataset which fluctuates daily with added noise.

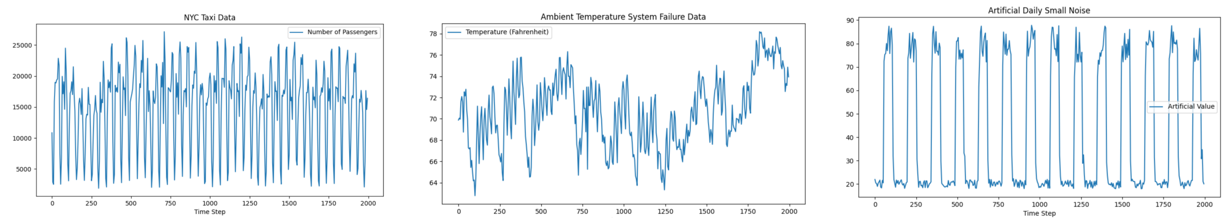


Figure 4.4: Graphs of our time-series datasets.

Some of these datasets are split into multiple parts, to be used by multiple tasks. Both nyc_taxi_1 and nyc_tax_2 originate from the Nyc Taxi dataset, but come from disjoint sections of it, and are treated as individual datasets for different tasks. The datasets used for each task have roughly 2,000 data points.

Task Name	**Time-Series Dataset**
affinity	nyc_taxi_1
atomic	nyc_taxi_2
bsearch	nyc_taxi_3
cap	nyc_taxi_4
chmod	art_daily_small_noise
memcpy	ambient_temperature_system_failure_1
vecmath	ambient_temperature_system_failure_2
zero	ambient_temperature_system_failure_3

Table 4.2: Each task and the corresponding time-series dataset used to model its workload fluctuations.

4.2.2 Forecasting Methodology

The time-series forecasting is done using LSTM neural networks from Keras [14][15]. For each task, we process the corresponding dataset and train the model using the following steps [16]:

1. Convert datasets into sequences of differences (where each value n_i, is converted into the difference $n_i - n_{i-1}$).

2. Standardize all values between -1 and 1.

3. Split the dataset into halves, with the first half used for training and the second half for testing.

4. Convert training and test sets into supervised learning datasets. For each n_i in the sequence, create a mapping that has n_i as the input and $(n_{i+1}, ..., n_{i+j})$ as the output, where j is the size of the forecasting window.

5. Train the LSTM model on the (input, output) values from the supervised training set.

6. Make predictions on the supervised test data.

7. Invert the transformations in (1) and (2) to get predictions for the actual dataset

For every point in each dataset, we now have the predicted values of the next j points, where j is the size of the forecasting window. In this research, we use a forecasting window of 20. Although larger forecasting windows result in more information, the prediction accuracy suffers if the forecasting window is too large.

4.2.3 Forecasting Error

In order to convert our predictions into actual workload sizes, we need to standardize them within our original range of $(10, 50)$. The same must be done with the real time-series data, which is given to tasks for their actual computations.

Then, we can evaluate the forecasts for each dataset using the RMSE for each prediction window. For each look-ahead window, j, $1 \leq j \leq 20$, we calculate the RMSE between our prediction of the value j steps in the future and the actual value j steps in the future. This look-ahead error is graphed for each dataset in Figure 4.5. Error begins relatively low, but for windows larger than 5, prediction error is much larger. As expected, prediction error increases when we attempt to predict further into the future.

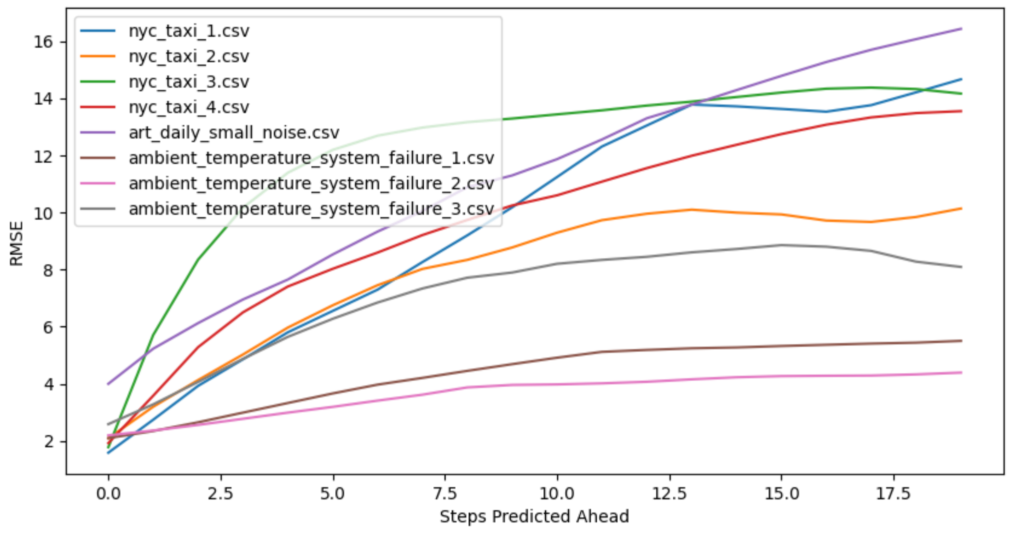


Figure 4.5: Prediction error for different look-ahead windows.

4.3 N-Step Resource Configuration Algorithm

At every time step, we need to decide whether or not to keep our current resource configuration. If we decide to keep the configuration, then can preserve our containers and continue onto the next step of the job. However, if we wish to change the configuration, we have to come up with a way to compute a new configuration. There are an exponential number of ways to divide up resources between all tasks in a job, so testing every combination is infeasible.

The optimal configuration at a given point in the job will be dependent on how long we plan to keep the configuration. A configuration that is optimized for a single time step may perform optimally for that time step, but it will suffer in performance beyond that.

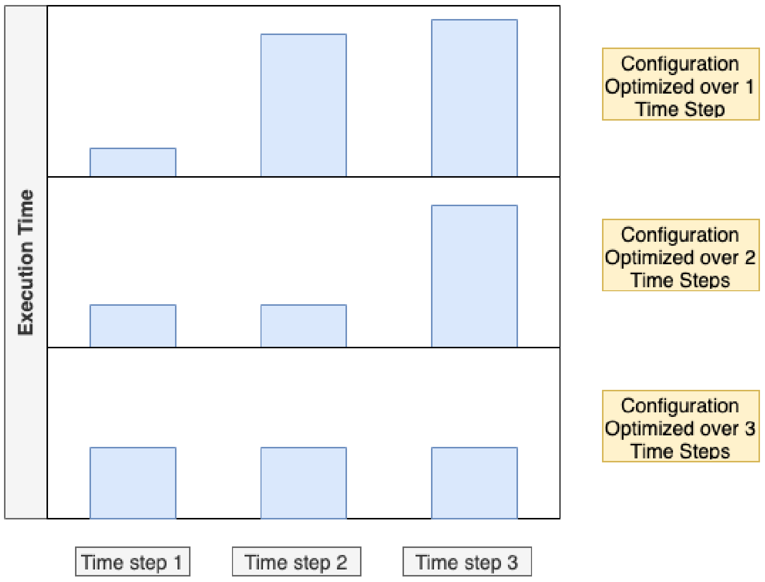


Figure 4.6: Performance comparison for resource configurations optimized over different time frames.

In Figure 4.6, we see that creating an optimal resource configuration for 1 time step results in the lowest duration for that time step out of all the configurations, but longer durations for later time steps. The resource configuration that is optimized for 3 time steps does not perform as well for the first time step, but it has the lowest total duration over all 3 time steps.

Although it may appear that creating a new configuration at each time step would minimize the total execution time, we need to consider the overhead associated with reconfiguration. This cost (the checkpoint penalty) discourages us from reconfiguring as often as possible. As discussed in Section 2.2.2, reconfiguration requires us to pause the job, checkpoint all tasks, and recreate containers on nodes with available resources. Therefore, it is better to keeping the current configuration if reconfiguration provides minimal improvement.

If we plan to keep a configuration for multiple time steps, we need to optimize it over that whole time-frame. This means, we want to minimize the cumulative duration over every time step in that range. An algorithm for creating such an optimal resource configuration would have to take into account the workload conditions at each time step.

The algorithm introduced in [2] can be used to choose resource configurations for static workloads. This thesis introduces an extension of that algorithm (Algorithm 1), which computes resource configurations for dynamic workloads over multiple time steps. The basis for the original algorithm is to begin with a default configuration and repeatedly add resources to the predicted slowest task. For multiple time steps with dynamic workloads, there may be a different “slowest task” at each time step. Like the original algorithm, we also begin with a default configuration (1 core per task). Then, we aggregate all tasks that are predicted to be the slowest at some time step (line 3 of Algorithm 1). Since these stragglers are forcing other tasks to wait, we need to provision more resources towards them. For each straggler, we use their predicted workload sizes and task performance model to estimate the cumulative improvement from incrementing their resource provisioning. We take the straggler that has the largest total improvement, and increment the resources for that task (line 9 of Algorithm 1).

This algorithm also prioritizes improvements in earlier time steps over later time steps. Since our predictions are subject to error, even if we plan to keep a resource configuration for *n* steps, that plan may need to be updated. Changes in our workload predictions could lead us to abandon our resource configuration earlier than anticipated. Therefore, we are more likely to keep our configuration for earlier time steps than later ones. The forecasting error also becomes increasingly inaccurate for predictions far into the future. This means that we can expect predictions at earlier time steps to be more accurate than later ones. These factors can be accounted for by applying a discount factor for “improvements” at later time steps.

Algorithm 1: Create Multi-step Resource Configuration

```
Input: predicted workloads, time frame
Output: Resource configuration optimized over that time frame
1 configuration ⟵ default
2 while resources available do
3     slowest jobs ← get slowest jobs(configuration, time frame, predicted workloads)
4     for time step ∈ time frame do
5         slowest job ← slowest jobs[time step]
6         improvements[slowest job]+= discount * improvement(slowest job, configuration, predicted
            workloads)
7     most improved job ← largest improvement(improvements)
8     increment configuration[most improved job]
9 return configuration
```

4.4 Reconfiguration Decision Planner

The last component of this research is the reconfiguration decision planner. At every time step, we can either keep the current configuration or generate a new configuration. If we decide to generate a new configuration, we have to decide how long we want to keep that configuration. Then, we can use Algorithm 1 to compute a new configuration for that time frame.

Assuming the longest time frame we intend to keep a configuration is n, then we have $n+1$ actions that we are able to take. These actions are $a_0, ..., a_n$, where we define a_0 as keeping the current configuration and a_i, $0 < i \leq n$, as creating a configuration that is optimized over the next i time steps. We can define a decision plan as a sequence of actions to take at each time step.

In this thesis, the reconfiguration decision planner is a pluggable component with multiple different implementations. New decision planning strategies can be tested by creating an implementation of the planner and observing its performance in our environment. In Chapter 5, we discuss the strategies used in this research for creating resource configuration plans. We will also evaluate the performance of each of these strategies using both forecasted predictions and “perfect” predictions.

CHAPTER 5

Resource Configuration Planning Algorithms

This chapter introduces three main resource configuration planning algorithms: a static approach, a dynamic approach (with a Model Predictive Control variation), and a reinforcement learning approach. Each algorithm knows our predictions for task workloads (up to the next 20 points), our current resource configuration, and performance models for each task. At each time step, the algorithm will output an action, from $a_0, ...a_n$, which informs the system on changes that must be made to our resource configuration.

5.1 Static Window

Our first strategy, the static window approach (Algorithm 2), is a naive approach which automatically updates the configuration every K steps, where K is the size of the static window. This new configuration is also optimized for the next K steps (using Algorithm 1), and kept for exactly K steps. This means that we take action a_k every K steps and action a_0 (keep the configuration) the remainder of the time.

The optimal size of this static window depends on the benefit of reconfiguration compared to the checkpoint cost. Larger checkpoint penalties bias towards larger static windows, but smaller checkpoint penalties bias towards smaller windows.

The static window approach provides some adaption to changes in task workloads. However, it will often make sub-optimal decisions. For example, it automatically discards the resource configuration at the end of the window, even if it is still performing well. Additionally, if our predictions are inaccurate, it can be stuck with a poor configuration that should be abandoned earlier.

Algorithm 2: Decision Planner for Static Windows

```
Input: Current time step, $t$, and window size, $w$
Output: Action to take
1 if $t$ % $w$ = 1 then
2     return $a_w$
3 else
4     return $a_0$ (keep current configuration)
```

5.2 Dynamic Algorithm

If our job consists of m total time steps and we can only keep a configuration until the end of the job, then we will have $m-t+2$ possible actions at each time step t. At each time step we can keep the current configuration

a_0 or create a configuration for anywhere between a single time step, a_1, to the end of the job, a_{m-t+1}. This leads to an exponential number of different reconfiguration plans.

In order to find the optimal plan of actions, we should try to eliminate sequences of actions that are clearly sub-optimal. In this approach, we make the assumption that if we take action $a_j,\ j \neq 0$, then we will always keep that configuration for exactly j time steps. This means that, in a configuration plan, action a_j will always be followed by the action a_0 exactly $j-1$ times. If we decide to keep the configuration for longer or shorter than j time steps, then it would have been preferable to create a configuration optimized over that time-frame instead. This constraint greatly reduces the number of total configuration plans that we need to consider.

5.2.1 Dynamic Programming

At time step t in the job, the goal is to find the action a_j which minimizes the duration of the job from time step t onwards. Let us first assume that we are at the start of the job, and we have no current configuration. Then, a_0 is not allowed, and we have to take actions $a_1, ..., a_{m-t+1}$ where we create a new configuration.

Then, we can define q_j as the minimum duration of the job starting from time step t, if we have to reconfigure at t (in other words we take an action other than a_0). The value we are trying to minimize is q_1 - the duration of the job starting from time step 1.

For a given action $a_j,\ 1 \leq j \leq m-t$, the computation time of the rest of the job is equal to:

$$compute_{t,j} = d_{t,t+j-1}(config_j) + q_{t+j} + C \tag{5.1}$$

Here, we define each of the following variables:

- $config_j$ - the configuration decided by a_j using Algorithm 1.
- $d_{t,t+j-1}(config_j)$ - the total execution time from time step t to time step $t+j-1$ inclusive using the configuration $config_j$.
- q_{t+j} - execution time of the job starting from a reconfiguration at time step $t+j$.
- C - the checkpoint penalty.

If we take the remaining action, a_{m-t+1}, then are creating a configuration for the remainder of the job. This would mean that there is no checkpoint penalty since this configuration does not need to be replaced. The duration of the job would simply be $d_{t,m}(config_{m-t+1})$.

$$compute_{t,m-t+1} = d_{t,m}(config_{m-t+1}) \tag{5.2}$$

This is the duration that the configuration ($config_{m-t+1}$) would take from time step t to the end of the job, time step m.

We can then set q_t as the minimum of all these possible durations

$$\begin{aligned} q_t &= min(\{compute_{t,j} \quad \forall j,\ 1 \le j \le m-t+1\}) \\ &= min(\{d_{t,t+j-1}(config_j) + q_{t+j} + C \quad \forall j,\ 1 \le j \le m-t\}, \quad d_{t,m}(config_{m-t+1})) \end{aligned} \tag{5.3}$$

The optimal action $a_{j_{opt}}$ is defined by:

$$j_{opt} = \underset{j}{\operatorname{argmin}}\,(compute_{t,j}) \quad \forall j,\ 1 \le j \le m-t+1 \tag{5.4}$$

Let us define the optimal action, j_{opt} (or $a_{j_{opt}}$) at time step t, to be j_t. In order to compute q_i and j_i $\forall i,\ 1 \le i \le m$, we should use equations 5.3 and 5.4. Since these equations are defined recursively, we can use dynamic programming to reuse computations. This can be done by computing backwards, starting at time step m to find q_m and j_m. At time step m, there is only one possible action available - a_1. The calculations for q_m and j_m are relatively simple and we can save the results in a map. Then, using our equations, we can work our way backwards to solve $q_{m-1}, ..., q_1$ and $j_{m-1}, ..., j_1$.

From these values we are able to create our optimal reconfiguration plan. When a job begins, there is no current configuration. The minimum duration of this job, using this algorithm, can be denoted as q_1. The first action in our configuration plan would be j_1 (or a_{j_1}). This means that we create a configuration for the next j_1 steps, and keep it (repeatedly take action a_0 afterwards) until time step j_1. Then, we create our next configuration for j_{j_1} steps (take action $a_{j_{j_1}}$). We can repeat this until we have reached time step m.

Algorithm 3: Dynamic Programming Decision Planner

```
Input: Workload predictions, $w$, and length of job, $m$
Output: Plan of actions
1 $qValues \longleftarrow$ List of zeros, length $m$
2 $jValues \longleftarrow$ List of zeros, length $m$
3 for $t \leftarrow m$ to 1 do
4     $qValues[t] \leftarrow$ Result from equation 5.3
5     $jValues[t] \leftarrow$ Result from equation 5.4
6 return computePlan(qValues, jValues)
```

5.2.2 Dynamic Example

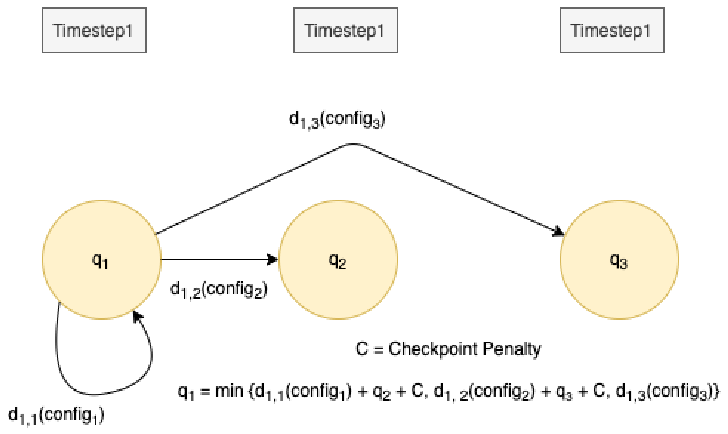


Figure 5.1: Example of using the dynamic programming solution on a Bulk Synchronous Parallel job.

Figure 5.1 shows an example of using the dynamic programming approach to calculate q_1 in a job with three time steps. Here, the values for q_2 and q_3 have already been calculated. There are three possible actions at time step 1 (a_1, a_2, a_3). Note that since time step 1 is the first step in the job, there is no current configuration and a_0 is not a possible action.

Here is the duration of the job, starting at q_1 for each of the actions:

- a_1 - We create a configuration for one time step and need to create a new configuration at time step 2. The duration for the job will consist of the execution time at time step 1 using the new configuration $(d_{1,1}(config_1))$, the computation time of the job starting with a reconfiguration at time step 2 (q_2), and the checkpoint penalty for reconfiguring at time step 2 (C).

- a_2 - We create a configuration for two time steps and need to create a new configuration at time step 3. The job duration consists of the computation time for time steps 1 and 2 using the new configuration $(d_{1,2}(config_2))$, the computation time of the job starting with a reconfiguration at time step 3 (q_3), and the checkpoint penalty for reconfiguring at step 3 (C).

- a_3 - We create a configuration for three time steps (or the entire job). No checkpoint is required as the configuration is never changed. The job duration consists of the computation time from time step 1 to 3 using the new configuration $(d_{1,3}(config_3))$.

We calculate q_1 by taking the minimum of these three values.

5.2.3 Model Predictive Control

The previous approach uses backtracking to compute our reconfiguration plan. Assuming that we knew the exact future workloads of each task, this solution would produce an optimal resource configuration plan. However, the resource configurations themselves may not be optimal because Algorithm 1 is not guaranteed to be optimal. This dynamic programming approach only ensures that if we use Algorithm 1 as our resource configuration algorithm, we supply the optimal sequence of actions to the algorithm.

Additionally, perfect knowledge of the future is unrealistic, and we often need to rely on imperfect model predictions. Since forecasting error increases for more distant predictions, as described in Figure 4.5, predictions beyond a certain point may be unreliable. Therefore, it is difficult to make meaningful configuration decisions too far into the future. This reduces the effectiveness of the dynamic programming approach because each decision is based off many future decisions.

Instead we can take a model predictive control approach for this problem (Algorithm 4). We will select a limited horizon (in this research, we use a horizon of 10 time steps), where forecasting predictions have stronger accuracy. Then, we will act as if the job ended at the end of the horizon. Instead of m being the last time step of the job, m will simply be the time step at the end of our window ($t+9$). We create a reconfiguration plan $(j_t, ..., j_{t+9})$ that optimizes the execution time over this horizon (time step t to time step $t+9$). Then, we will take the first action, j_t of the plan. When we arrive at our next time step, $t+1$, we will have updated predictions and our horizon will be 1 time step further into the future ($t+1$ to $t+10$). Then, we can recompute our configuration plan, and again, take the first action of that plan. At time step t, when we are computing q_m and j_m at the end of the window, we are now actually computing q_{t+9} and j_{t+9}. We can work our way backwards until we get q_t and j_t.

However, the value q_t makes the assumption that we have to reconfigure at time step t. In this model predictive control variation, it is possible that at the beginning of our window, time step t, we have a current configuration, $config_c$. Therefore, we have the option of taking action a_0. Let q'_t be the minimum duration of the job from time step t if we keep the current configuration, $config_c$ (or in other words, take a_0).

$$q'_t = min(\{d_{t,t+j-1}(config_c) + q_{t+j} + C, \forall j, 1 \leq j \leq m-t\}, d_{t,m}(config_c)) \tag{5.5}$$

j represents how long we keep the current configuration, $config_c$. This could be anywhere from 1 time step to $m-t+1$ time steps. If $q'_t < q_t + C$, then we know that we can get better performance by keeping our current configuration than creating a new configuration at time step t. Therefore, at time step t, the first action in our plan (and also the action we should take) would be a_0 instead of a_{j_t}.

Algorithm 4: Model Predictive Control Variation of Dynamic Algorithm

```
Input: Workload predictions, $w$, current time step, $t$, end of window, $m$
Output: Action to take
1  $C \longleftarrow$ Checkpoint penalty
2  $qValues \longleftarrow$ List of zeros, length $m - t + 1$
3  $jValues \longleftarrow$ List of zeros, length $m - t + 1$
4  for $t' \leftarrow m$ to $t$ do
5      $qValues[t'] \leftarrow$ Result from equation 5.3
6      $jValues[t'] \leftarrow$ Result from equation 5.4
7  $q'_t \leftarrow$ Result from equation 5.5
8  if $q'_t < q_t + C$ then
9      return $a_0$
10 else
11     return $a_{jValues[t]}$
```

5.3 Reinforcement Learning Approach

If we know the future workload sizes for each task, then the dynamic programming approach in Section 5.2 should find the optimal sequence of actions for our reconfiguration plan. Note, that it may not find the optimal configurations themselves, as Algorithm 1 is greedy and may not be optimal. The dynamic programming strategy should find the optimal sequence of actions if we decide to use Algorithm 1 to compute resource configurations.

However, that approach is dependant on our workload forecasting ability. If we have perfect knowledge of future workloads, then the dynamic approach can be used to compute the entirity of our configuration plan. It suffers when dealing with forecasting error, like the problem in this thesis work. Even though the model predictive control adjustment in Section 5.2.3 allows the algorithm to work with forecasting predictions, it does not take into account that predictions at some time steps will have higher errors than others. This can partially be dealt with by discounting longer configuration windows (which rely on more distant predictions), but this does not allow the model to learn from the actual forecasting error at each step.

This seems to be a problem space where reinforcement learning can be applied. There is a discrete set of actions, $\{a_0, ..., a_n\}$ and a very well defined reward function (lower computation time). Additionally, reinforcement learning might be able to learn from forecasting error and estimate the trustworthiness of each prediction. This would allow the agent to make decisions based purely off strong predictions.

For our reinforcement learning agent, we implemented an environment using OpenAI Gym and used a Deep Q Learning agent from the stable baselines package [17][18][19].

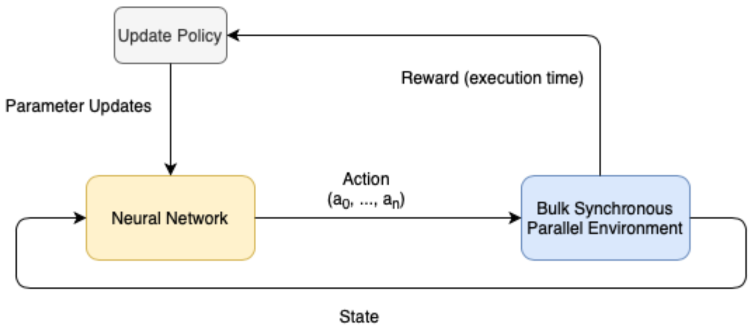


Figure 5.2: System for training the Deep Q Learning agent. The neural network accepts a state and computes an action to take. The action is fed into the environment, which returns the next state and the execution time (which is used as the reward for training).

Initially, we hoped that we could use predicted workloads of each task as the observation, have the action be one of $\{a_0, .., a_n\}$, and have the reward be based on its execution time.

However, this failed to produce any meaningful results. It may have been difficult for the agent to map from predicted workloads to task duration and realize that the slowest task dictated the overall job performance.

Instead, this thesis sets the state as an $n+1$ by m matrix, where n is the longest we are willing to keep a configuration and m is the number of time steps that we are looking ahead. We have $n+1$ possible actions $(a_0, ..., a_n)$ and each row corresponds to an action. Then, in this matrix, the value at $(x$, $y)$ - in the row x and column y is the predicted duration y time steps from now with the configuration $config_{x-1}$. $config_{x-1}$ is a configuration that is optimized for the next $x-1$ time steps, or resulting from action a_{x-1}.

For example, let us take the element at row 1, column 5 of a state. Elements in row 1 correspond to the configuration resulting from action a_0, or the current configuration. Column 5 means we are looking at the execution time 5 time steps from now. Therefore, this element represents the execution time 5 time steps in the future using the current configuration ($config_c$).

CHAPTER 6

Experiment

6.1 Experimental Setup

Our experimental setup uses a bare-metal Chameleon Cloud instance [20]. We create four virtual machines on the instance which act as the nodes of our Kubernetes cluster. The specifications of our cluster can be found on Table 6.1.

Kubernetes Cluster Specifications	
Number of Nodes	4
CPU Cores per Node	10vCPU
Memory per Node	10GB
Operating System	Ubuntu 20.04
Kubernetes Distribution	Microk8s

Table 6.1: Specifications for the Kubernetes cluster used in these experiments.

Our tasks were run as Kubernetes jobs using custom Docker containers [5]. Resource requirements and benchmark specific parameters were passed in as a part of the job template. The Bulk Synchronous Parallel model was enforced using zookeeper [21]. All tasks had to enter the zookeeper barrier before computation could begin, and all had to leave the barrier for the time step to end. Workload sizes are passed to individual tasks using queues in zookeeper.

Although the code is in place to run these benchmarks as a real Bulk Synchronous Parallel job, the results in this section are obtained through simulation. Each time step can take several minutes to complete and we want to test our configuration strategies on a large number of time steps. This would lead to many hours of computation for each experiment. Therefore, we use the task performance models from Section 4.1.3 to estimate task run-times. This research makes the assumption that our profiling data is relatively accurate. Instead of running the benchmarks and measuring the durations, we use the task performance models to estimate the execution time of each task. This allows us to test our reconfiguration strategies on large numbers of time steps in a much shorter time-frame.

Each of our reconfiguration strategies are tested on a simulated Bulk Synchronous Parallel job consisting of the 8 stress-ng benchmarks described in Table 4.1. The job is simulated for 400 time steps and task workloads are modelled by the corresponding datasets in Table 4.2. The checkpoint penalty, as described in Section 2.2.2 is configurable, but for these experiments has been set to 15 seconds.

6.1.1 Evaluated Strategies

We evaluated the following types of strategies:

- **EXPPO** - This uses the approach from [2], in which we assume that task workloads are static. We assume average-case workloads for each task and then create an optimal configuration for those workloads. This configuration is used at every time step of our simulated job.
- **Static Window - N** - This uses the static approach from Section 5.1, where a new configuration is created every N steps. In this experiment, we tested static windows from size 1 (reconfigure every step) to size 9. The optimal value for N will vary based on the size of the checkpoint penalty.
- **Dynamic Model Predictive Control** - This strategy uses the Model Predictive Control variation (Section 5.2.3) of the dynamic algorithm. It uses a maximum horizon of 10 time steps.

6.2 Results with Synthetic Prediction Data

As mentioned in Chapter 5, many of these reconfiguration strategies are influenced by forecasting error. We want to first evaluate these configuration strategies in an environment where future workloads are known. These results are produced by substituting predicted workloads with the real workload values and testing each strategy. If a strategy performs poorly without any forecasting error, it is unlikely that it will perform well with imperfect workload predictions.

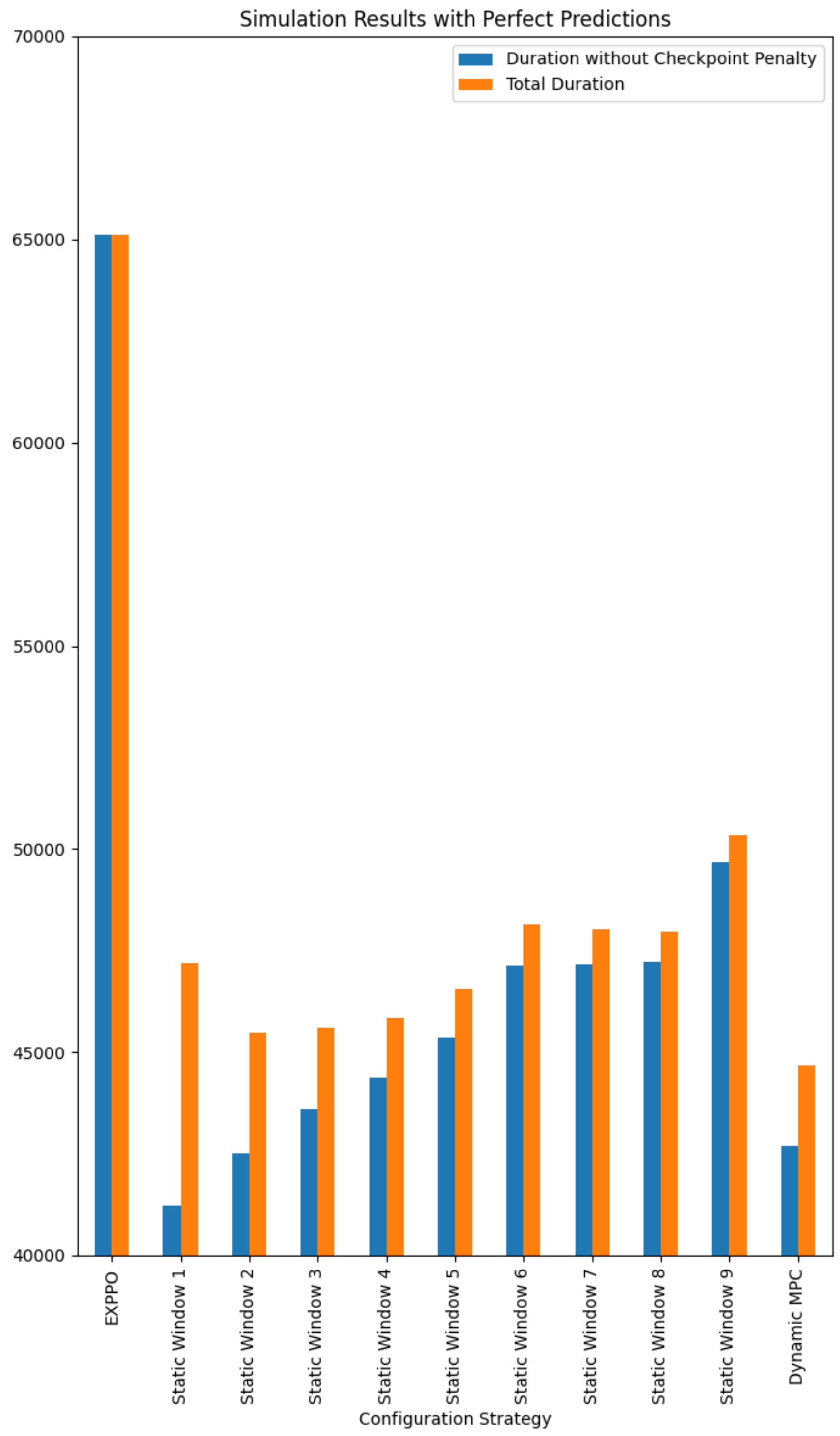


Figure 6.1: Simulation results for each reconfiguration strategy assuming perfect knowledge of future workloads. The execution time with and without reconfiguration (checkpoint) penalty is shown.

The execution time of each strategy is shown in Figure 6.1. As expected, the Static Window 1 strategy produces the best result when we do not account for reconfiguration cost. This is the strategy where we create a new configuration at every time step.

The Dynamic Model Predictive Control strategy performed the best when accounting for reconfiguration cost. It's execution time was 816 seconds faster than the best Static Window strategy (Table 6.2). All strategies outperformed the EXPPO strategy, which never changed the configuration.

Configuration Strategy	Execution Time without Checkpoint Penalty	Number of Checkpoints	Total Execution Time
EXPPO	65126.48343786634	0	65126.48343786634
Static Window 1	41221.57927906762	399	47206.57927906762
Static Window 2	42508.68648937608	199	45493.68648937608
Static Window 3	43606.13326437778	133	45601.13326437778
Static Window 4	44366.02578712543	99	45851.02578712543
Static Window 5	45375.0902513685	79	46560.0902513685
Static Window 6	47146.89345392543	66	48136.89345392543
Static Window 7	47173.10259364447	57	48028.10259364447
Static Window 8	47223.891749660346	49	47958.891749660346
Static Window 9	49673.6481354634	44	50333.6481354634
Dynamic MPC	42697.6013299094	132	44677.6013299094

Table 6.2: Simulation results for each reconfiguration strategy assuming perfect knowledge of future workloads. Lists the execution time with and without reconfiguration (checkpoint) penalty as well as the number of checkpoints.

6.3 Results using Model Predictions

We then evaluate the same strategies using our forecasting predictions. Each of the strategies now has to make decisions based on predicted workloads. These results showcase how the strategies handle forecasting error.

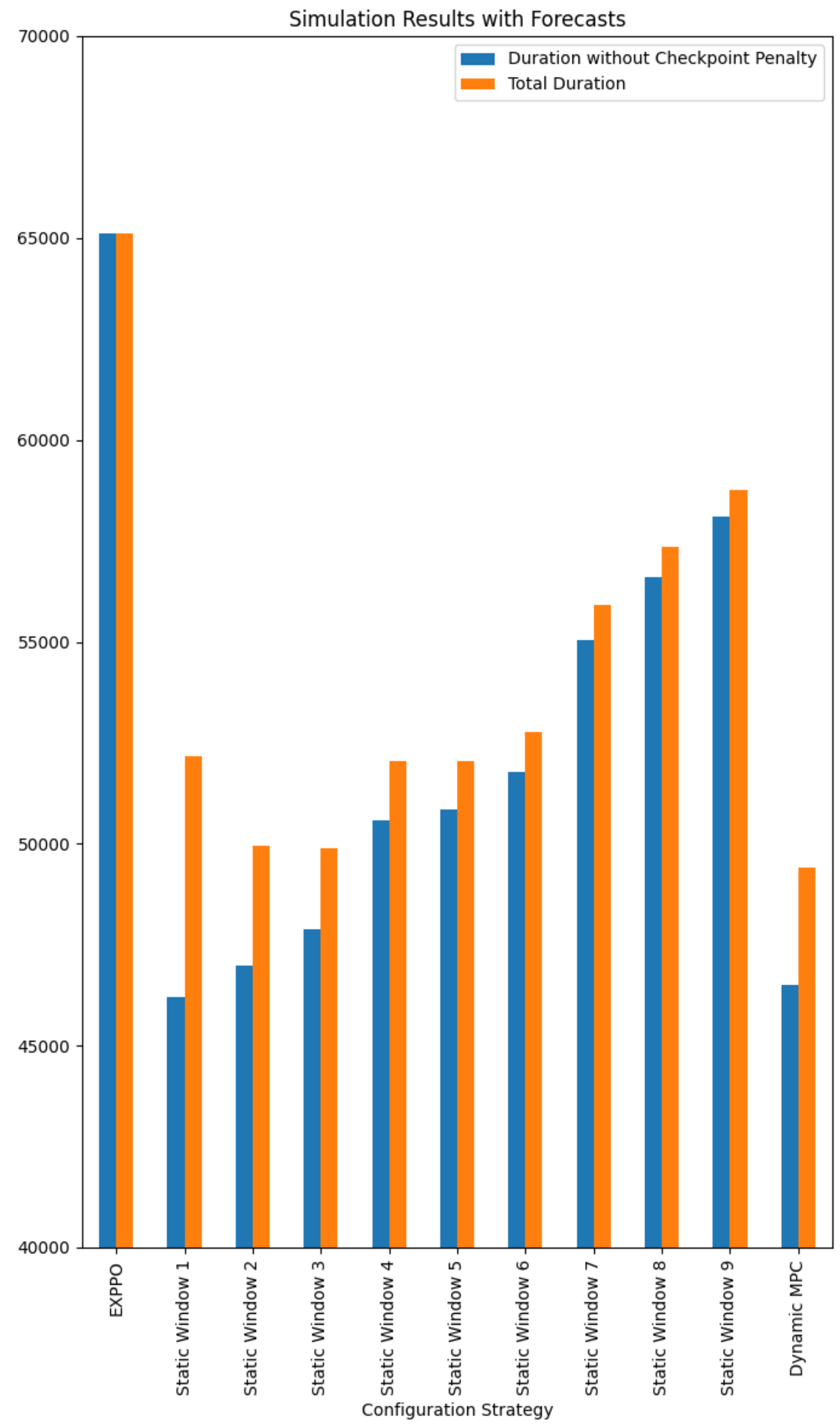


Figure 6.2: Simulation results for each reconfiguration strategy using forecasts as workload predictions. The execution time with and without reconfiguration (checkpoint) penalty is shown.

The results of these simulations are displayed in Figure 6.2 and Table 6.3. Once again, all strategies outperformed the EXPPO strategy, which never adapted the configuration. However, the non-EXPPO strategies all performed worse without perfect workload predictions. In this experiment, the Dynamic MPC strategy was only 467 seconds faster than the strongest Static Window approach. It also reconfigured much more often than in the previous experiment, (194 vs. 132 times), showing that it had to frequently abandon poor configurations that resulted from inaccurate predictions.

Configuration Strategy	Execution Time without Checkpoint Penalty	Number of Checkpoints	Total Execution Time
EXPPO	65126.48343786634	0	65126.48343786634
Static Window 1	46195.27997458069	399	52180.27997458069
Static Window 2	46970.590944808864	199	49955.590944808864
Static Window 3	47880.17876125451	133	49875.17876125451
Static Window 4	50574.74648368374	99	52059.74648368374
Static Window 5	50863.57542578313	79	52048.57542578313
Static Window 6	51787.42560532594	66	52777.42560532594
Static Window 7	55054.509896360505	57	55909.509896360505
Static Window 8	56611.79980463157	49	57346.79980463157
Static Window 9	58099.97635820614	44	58759.97635820614
Dynamic MPC	46497.925205953434	194	49407.925205953434

Table 6.3: Simulation results for each reconfiguration strategy using forecasts as workload predictions. Lists the execution time with and without reconfiguration (checkpoint) penalty as well as the number of checkpoints.

6.3.1 Results with Reinforcement Learning

This thesis also evaluated the reinforcement learning strategy with forecasted workloads. It was able to perform 159 seconds faster than the dynamic strategy (Table 6.4).

However, in order to get the model to converge, it was trained on a shorter prediction window (6 time steps) and a smaller set of actions (a_0, a_1, a_2, a_3). This means that the reinforcement learning strategy would not attempt to create a configuration that spanned more than three time steps. These factors may have contributed to this strategy being less vulnerable to prediction error.

Configuration Strategy	Execution Time without Checkpoint Penalty	Number of Checkpoints	Total Execution Time
Deep Q Learning	46938.78736588008	154	49248.78736588008

Table 6.4: The same simulation results as Table 6.3 except with the Deep Q Learning strategy.

6.4 Summary of Results

These results show the benefit of adapting our resource configurations to workload fluctuations. All of the other strategies performed better than the EXPPO strategy, which kept a single resource configuration. This improvement is more significant in scenarios where we have perfect knowledge of future workloads. In particular, the dynamic strategy, from Section 5.2, has the strongest performance with this information.

However, the performance of the dynamic strategy is greatly reduced when we are making decisions based off of less accurate predictions. In these situations, where prediction uncertainty is high, the reinforcement learning strategy from Section 5.3 can be a strong alternative.

When using the dynamic strategy, we must also consider the overhead of the algorithm itself. For longer running jobs, where performance benefits may be higher, the dynamic algorithm may reduce overall execution time. However, for faster running jobs, the overhead associated with the algorithm may be more than the actual performance benefits it provides.

CHAPTER 7

Conclusion

In the Bulk Synchronous Parallel model, computation performance can be greatly improved by reducing the execution time of stragglers. Allocating more resources towards larger computations can reduce the wait time of all other tasks. Previous research provides the groundwork for allocating resources for Bulk Synchronous Parallel tasks. However, it relies on the assumption that each task performs the same computation at every time step.

This thesis extends that work by accounting for dynamic workloads for each task. It combines workload profiling and time-series models to make predictions about future performance. It also introduces algorithms for choosing resource configurations and deciding how long to keep a configuration. Finally, this research provides a framework for evaluating our reconfiguration decision planning algorithms in both a simulated and real Kubernetes environment.

7.1 Future Work

There are multiple areas in which we hope to make further improvements in this research:

- **Memory Considerations** - In this thesis, we focus solely on CPU for our resource configuration. However, some tasks may benefit more from increasing memory than CPU shares. By adding an additional dimension to our resource configuration, we could better accommodate these types of tasks. This would allow us to provide a more detailed resource configuration and adapt this approach to a wider range of tasks.

- **Compare with Reactive Approach** - This research takes a proactive approach towards resource scaling, by using predicted workloads for each task. However, we may not be able to get accurate predictions for certain types of tasks. Future work could instead explore a reactive approach, which responds to changes in workload sizes.

- **Evaluate In-Place Pod Updates** - Only one type of reconfiguration is considered in this research: creating a new configuration from scratch, which requires all containers to be checkpointed and recreated. However, there are other ways to adjust the resource configuration that do not incur a checkpoint penalty. If multiple tasks are scheduled onto the same virtual machine, resources may be passed between these tasks without needing to delete any containers. Future work can compare this type of reconfiguration with creating a new configuration from scratch. At the time of this thesis, Kubernetes does not yet support in-place resource updates, but it is being actively developed.

- **Account for Vertical Interference** - Some tasks may run into interference issues if they are scheduled onto the same node. They may contend for the same shared resources, which would result in performance degradation. Future work can conduct more extensive task profiling to discover which combinations of tasks will experience interference. This could lead to scheduling configurations which inform the cluster about tasks to avoid scheduling together.

References

[1] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing-a paradigm for transportable software. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 268–275 vol.2, 1995. doi: 10.1109/HICSS.1995.375451.

[2] Yogesh D. Barve, Himanshu Neema, Zhuangwei Kang, Harsh Vardhan, Hongyang Sun, and Aniruddha Gokhale. Exppo: Execution performance profiling and optimization for cps co-simulation-as-a-service. *Journal of Systems Architecture*, 118:102189, 2021. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2021.102189. URL https://www.sciencedirect.com/science/article/pii/S138376212100134X.

[3] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[4] Karen E. Willcox. Predictive digital twins: Where dynamic data-driven learning meets physics-based modeling. DDDAS2020 Conference, 2020. URL https://s3.amazonaws.com/static.1dddas.org/docs/2020ISC/presentations/Keynote-Willcox.pdf.

[5] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[6] Kubernetes. https://kubernetes.io/, 2014.

[7] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.

[8] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507, 2011. doi: 10.1109/CLOUD.2011.42.

[9] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. A control-based framework for self-managing distributed computing systems. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, WOSS '04, page 3–7, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139896. doi: 10.1145/1075405.1075406. URL https://doi.org/10.1145/1075405.1075406.

[10] Colin Ian King. Stress-ng. *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017.

[11] Kubernetes documentation. https://kubernetes.io/docs/home/, 2021.

[12] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[13] Alexander Lavin and Subutai Ahmad. Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 38–44. IEEE, 2015.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] François Chollet et al. Keras. https://keras.io, 2015.

[16] Jason Brownlee. Multistep time series forecasting with lstms in python. https://machinelearningmastery.com/multi-step-time-series-forecasting-long-short-term-memory-networks-python/, 2017.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[19] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[20] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.