Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Energy Efficient Portfolio Scheduling for Business-Critical Workloads

Author: Timo van Milligen (2684444)

1st supervisor:	Alexandru Iosup
daily supervisor:	Vincent van Beek
2nd reader:	Animesh Trivedi

A thesis submitted in fulfillment of the requirements for

the joint UvA-VU Master of Science degree in Computer Science

September 27, 2022

Abstract

Cloud service providers consume an extremely large amount of electricity in their datacenters, resulting in high operational costs and carbon footprints. As energy prices soar, energy consumption in datacenters become an increasing risk to the profit margins of datacenters, and energy efficiency becomes increasingly important. While most of the effort to improve energy efficiency has gone into improving the infrastructure, like cooling, energy efficiency can also be improved on the server level, with energy efficient virtual machine (VM) placement.

Our approach on energy efficient VM placement is based on portfolio scheduling, which is a scheduling technique that dynamically selects a scheduling policy from a set of policies, based on a utility function. We extend previous work in portfolio schedulers with a reflection stage, which uses historic performance of the portfolio to suggest policies to be added to the portfolio, to likely improve its performance. Our work is the first to implement the reflection stage of a portfolio scheduler capable of reflecting on the portfolio. Our work is also the first to use portfolio scheduling in the context of energy efficient scheduling.

We implement and evaluate a prototype of the designed portfolio scheduler to demonstrate its capabilities in energy efficient scheduling and of the reflection stage to adapt the portfolio based on past performance. We focus in this work on business-critical workloads representative for workloads of private Dutch cloud-service providers. Our experiments show that the designed reflection stage is capable of suggesting policies to be added to the portfolio that are likely to be selected in future scheduling decisions, but also that the energy efficiency of a scheduling decision for business-critical workloads in the short term is not always representative for the efficiency in the long-run, and that this can lead to sub-optimal policy selections for a portfolio scheduler. We also find that an energy efficient utility function for the portfolio scheduler can significantly impact other performance metrics, signalling the need for a more balanced utility function. With this work we have taken the first step towards a general portfolio scheduler capable of reflecting and adapting by itself based on past performance.

Contents

1 Introduction

Demand for data centers keeps increasing. In 2018 datacenters were already using an estimated 200Twh/year, or about 1% of the global energy demand at the time [1]. This demand is expected to keep growing, with an estimated tripling of cloud compute instances and workloads between 2016 and 2021 [2], and no signs of slowing down. Some studies suggest the energy demand of datacenters could quadruple by 2030 [3]. More efficient servers along with more efficient scheduling and virtualization helps dampen infrastructure energy use when compute instances increase. In the past this has enabled a six-fold increase in compute instances while resulting in only a 25% increase in global server energy use [2]. Not only can more efficient scheduling save energy and therefore money, it can also improve user-impacting performance such as job slowdown. Although refined scheduling policies exist that address specific workload characteristics, types of applications and utility functions, data centers still use human system administrators to choose a scheduling policy and configure it properly. A shift in performance goals or change in workload characteristics can however render a scheduler ineffective. Furthermore, previous studies have shown that no single scheduler consistently performs better than all others for complex workloads [4][5]. Portfolio scheduling aims to tackle these issues by dynamically switching between a portfolio of scheduling policies.

Previous studies in portfolio scheduling for distributed systems have already shown that this process can lead to meaningful solutions for different workloads and environments [6][7][8][9].

The creation and reflection of a portfolio is common practice in other fields. In finance a portfolio of assets can be methodically constructed to balance a portfolio's risk and rewards [10]. This concept has also been adopted in certain areas of computer science, like for constructing a portfolio representing a virtual cloud cluster, made up of different transient server types [11]. Given an application's tolerance to revocation risk it can construct an appropriate portfolio of server types, each with their own risk and reward similar to how a financial portfolio of assets is constructed. The implementation of portfolio scheduling in distributed systems as defined in this thesis *does* differ from the implementation of portfolios in the previously mentioned areas. The previous examples result in a portfolio of assets, which are all to be used simultaneously, while portfolio scheduling creates a portfolio of policies, but only picks a single policy to be applied at once. The risk and reward configuration of the portfolio scheduler therefore exists mostly in the selection of the policy to be applied instead of in the construction of the portfolio itself, as is the case in other fields.

1.1 Context

Datacenters usually employ of virtualization techniques to divide resources of a single machine over numerous virtual machines (VMs) [12]. Virtualization installs a hypervisor on top of a physical machine, and install a number of virtual machines on top of the hypervisor, each with their isolated environment. This is

in contrast with the traditional approach of installing a single operating system on top of a physical machine, and running an application on the operating system. Figure 1 depicts the difference in these architectures. Because resource utilization in datacenters is often below 15% [13], virtualization can consolidate multiple virtual machines on a single machine and improve resource utilization.

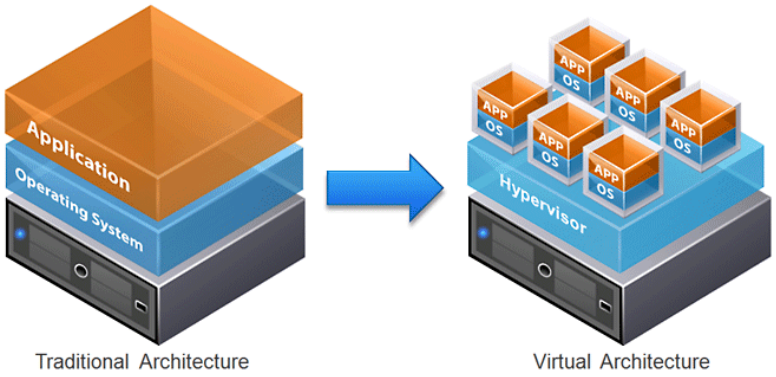

Figure 1: Traditional vs virtual architecture common in datacenters Source: [14]

In cloud datacenters running business-critical workloads, the workload is expressed as requests to schedule (often long running) virtual machines for customers to run their business-critical workloads in, instead of requests to execute a specific job or application. Scheduling of virtual machines to physical machines is a non-trivial task, because of the possible multi-objective (performance and resource utilization), and multi-dimensionality of the problem in terms of CPU utilization, memory utilization, disk space, network load, IO load and CPU cores. In addition to this that the CPU and memory workloads are relatively unpredictable for business-critical workloads [6] in comparison to workloads from for example Google [15].

1.2 Problem Statement

Dang et. al [8] describe in their work the four stages of a fully equipped portfolio scheduler: *creation* of the portfolio, *selection* of policies, *apply* the selected policy and *reflect* on the constructed portfolio. Previous studies have focused on either only the first three stages [6][9][6], or on using historical data to adapt the selection process [7]. We identify a few problems with previous work in portfolio scheduling:

First, setting up a portfolio scheduler requires non trivial adaptation to workload and environment. This results in essentially the same, and possibly even bigger problems that come with the configuration of a traditional scheduler in data centers. Instead of choosing and configuring a single scheduling policy, a human system administrator would now have to select and configure a

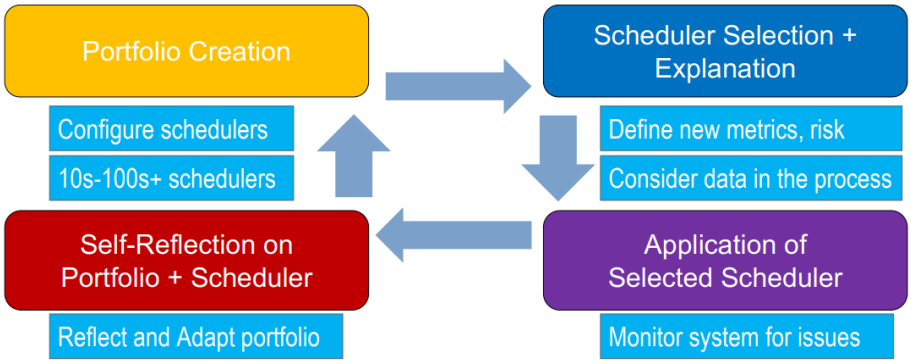

Figure 2: The four stages of a generalized portfolio scheduler.

whole portfolio of scheduling policies, introducing even more complexity to this initialization process.

Second, it lacks the ability to use historical data to improve future scheduling. While a previous study in portfolio scheduling has used historical data in the form of reinforcement learning, this is only done for industrial workloads that show periodic patterns [7]. Also, this reinforcement learning only affects which policy to select and does not adapt the selected portfolio of policies. This prevents the portfolio scheduler from automatically adapting to future changes in workload patterns for which policies outside of the selected ones might perform better.

Third, as computing power is the major contributor to datacenter power consumption [16], and global datacenters are the second fastest growing in terms of global greenhouse gas emissions from the whole ICT sector [17], we observe the need for more energy efficient VM placement. More efficient VM scheduling could significantly improve performance to power ratios and improve energy savings for datacenter operators.

1.3 Research questions

The main goals of this thesis are to make portfolio scheduling more generally capable and to create a portfolio scheduler that optimizes for energy efficiency. We divide the problem of generally capable energy efficient portfolio scheduling in two research questions, each with two sub questions:

(RQ1) Design: How can we design a more generally capable energy aware portfolio scheduler?

There is a high level of complexity for human system administrators in constructing a portfolio of policies, and in maintaining a portfolio scheduler. Answering this research question can help reduce this complexity. This is challenging, because we need to design from scratch a way to adapt part of the portfolio scheduler based on a given workload . We pose the following sub-questions to

answer this question:

(RQ1.1) How can we design a portfolio scheduler to dynamically reflect on the equipped portfolio?

A reflection stage of the portfolio scheduler capable of suggesting adaptations to the portfolio that are likely to be selected will make it more general, as it suggest adaptations based on the historic workload and portfolio performance, reducing complexity for human system administrators in reflecting on the scheduler and adapting it to changes in workload characteristics. This is challenging because it is unexplored territory and requires novel designs. Since portfolio scheduling is still a relatively new concept, previous work has so far only used historic data to adapt the selection stage [7], while the rest of the stages are static.

(RQ1.2) How can we design a portfolio scheduler to optimize for energy efficiency?

Considering the success of previous work in portfolio scheduling to improve over individual policies for numerous utility functions, adapting the utility function to select on energy-efficiency could lead to more energy-efficient scheduling of VMs. Previous work has not researched energy efficient scheduling for business-critical workloads, or with the use of a portfolio scheduler. Futhermore, in related work, a placement is often selected on the smallest increase in estimated energy consumption [18, 19], which can differ from how energy efficient a placement is based on the performance due to things like interference.

(RQ2) Evaluation: How can we evaluate a reflective energy aware portfolio scheduler?

To understand the operation of the designed portfolio scheduler, we need to quantify its performance. Conducting a sound evaluation is difficult, as it presents challenges in reproducibility and design. We pose the following subquestions to address this question:

(RQ2.1) How can we implement a prototype according to the designed portfolio scheduler?

In order to evaluate the performance, we need to implement a prototype of the designed scheduler.

(RQ2.2) How can we design, conduct, and analyze trace-based simulation experiments modeling cloud infrastructure?

Through simulation of cloud infrastructure we can analyse complex situations and different schedulers in a matter of minutes.

Closest related work to these research questions and specifically **RQ1.2** is an implementation of a portfolio scheduler for managing disaster-recovery risks for datacenters hosting business-critical workloads (BCWs) [20]. In BCWs, jobs are expressed as requests for virtual machines instead of requests to execute a specific application.

1.4 Approach

To address **RQ1**, we extend previous portfolio scheduler implementations with the ability reflect on and change the portfolio of policies, and the ability to optimize for energy efficiency. To address **RQ1.1**, we design a genetic search algorithm for scheduling policies that can make use of historic data in the form of system snapshots of past selection moments for the portfolio scheduler. Addressing **RQ1.2**, we extend the portfolio scheduler with the ability to optimize for an energy efficiency metric. We extend OpenDC, an open-source datacenter simulator [21], with an implementation of a portfolio scheduler. This ensures future users of the portfolio scheduler automatically benefit from potential improvements to the OpenDC platform. This process is guided by the iterative AtLarge design process [22].

To address **RQ2.1** we implement a working prototype on top of the OpenDC platform [21] that fulfills the requirements in the design resulting from **RQ1**. We then address **RQ2.2** by running trace-based experiments on the prototype using real-world business-critical workload traces. We then do performance evaluation of the portfolio scheduler in a multitude of dimensions, with the main focus being energy efficiency. We compare this against a baseline of single scheduling policies, and again after extending the portfolio scheduler by running genetic search on a collection of system state snapshots of past selection moments.

The experiments and the extension to OpenDC are released as free and open-source on GitHub[1].

1.5 Contributions

This thesis has resulted in the following contributions:

1. **(Conceptual)**.
   (a) **Design of general portfolio scheduler**: The contribution here is a novel design for a portfolio scheduler that uses historic system state snapshots in combination with a genetic search algorithm to reflect on its past performance. A reflection stage capable of adapting the portfolio could make the portfolio scheduler more resistant to changes in workload characteristics by finding policies outside of the selected ones that might perform better in the future.
   (b) **Analysis of the portfolio scheduler**: using experiments, we have evaluated and analyzed the energy efficiency optimizing capabilities of a portfolio scheduler that can utilize historic data.
   (c) **Analysis of energy efficiency optimization algorithm using genetic search**: By the means of experiments we have analyzed the optimization capabilities of a genetic search algorithm which tries to extend a portfolio scheduler to improve energy efficiency.

[1] https://github.com/Timovanmilligen/opendc

2. **(Technical)** Development and experiments

   (a) Experiment design, setup and validation of the portfolio scheduler for energy efficient VM placement.

   (b) Extension of the OpenDC simulator platform with a portfolio scheduler and system snapshot writer and loader.

   (c) To address reproducibility of the experiments, we release all the extensions to OpenDC done in this work as well as the experiments and all necessary data to run them as open-source on Github[2].

1.6 Statement of Ethical Development of the Thesis

I confirm that this thesis work is my own work, is not copied from any other source, and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

This thesis is structured as follows: In Chapter 2 we give an overview of subjects related to energy efficient scheduling in datacenter and portfolio scheduling. In Chapter 3 we present the design of the proposed Portfolio Scheduler. In Chapter 4 we present the evaluation of the Portfolio Scheduler through experiments and in Chapter 5 we present the conclusion and possible future work.

[2]https://github.com/Timovanmilligen/opendc

2 Background

We present in this chapter an overview of subjects related to the problem of energy efficient scheduling in datacenters and portfolio scheduling.

2.1 Energy Efficiency in Datacenters

A datacenter consumes on average as much energy as 25,000 households [13]. With the large carbon footprint of datacenters and the rising energy prices, there is an increasing need for energy efficient datacenters. A risk analysis as done by Radice [23], finds that rising electricity expenses have become the primary risk for datacenters in 2021. It finds that a price increase for electricy of 10x can impact the monthly expenses of datacenter operators by a factor of 60x-140x, especially for datacenters with a low power usage effectiveness. This price increase could impact the profit margin for cloud service providers, signalling the need for more energy efficient datacenters. Power usage effectiveness (PUE), proposed by the Green grid consortium is the prevailing metric for expressing datacenter energy efficiency. It is the ratio of the total amount of energy used by a datacenter, to the energy used by computing systems. However, PUE tells very little about the energy efficiency of individual components in a datacenter, Reddy et. al [24] have explored different metrics for energy efficiency. These metrics range from facility level, all the way to server level.

2.2 Energy Efficient Scheduling

One level on which energy efficiency can be increased is on the server level. Energy efficient resource allocation can significantly impact the energy consumption of servers. The difference between an optimal CPU utilization in terms of operations done per unit of energy consumed and a commonly found CPU utilization of below 15% in datacenters can be a factor of 4x [13, 25]. Energy efficient resoure allocation needs to consider the trade-off in power and performance, as consolidating multiple virtual machines on a single machine can lead to performance interference. Requirements regarding performance are defined in service level agreements (SLAs) between a cloud service provider and customers. Breaking these SLAs can lead to refunds, and reduced levels of customer satisfaction for cloud service providers.

To increase energy efficiency on the server level Cardosa et. al [26] use the functionality of virtualization software to specify the minimum and maximum amount of resources that can be allocated to a VM, to ensure intelligent distribution of resources in their designed VM placement technique. However, this requires knowledge of the application priorities to configure the parameters. For business-critical workloads, such details about the software are often too sensitive to reveal.

Beloglazov et, al [18] introduce a modified best fit decreasing (MBFD) policy for VM provisioning, which estimates the increase in power consumption by placing a VM on a host and selects the host with the smallest increase in power

consumption. On its own this policy does fail to consider the performance (and therefore efficiency) that can be achieved by the VM with the evaluated placement, as it only considers the increase in power. The proposed solution does include a VM migration policy to manage bad provisioning choices based on a threshold in CPU utilization.

In this work we focus solely on the energy efficient placement of VMs.

2.3 Portfolio Scheduling

A portfolio scheduler consists of a portfolio of policies, of which only one is applied at a specific point in time. While a regular scheduler consists of a single (though sometimes quite complex [27]) policy of where a workload should go, these are often specialized for specific environments and workload characteristics. When the workload pattern or the environment changes, there is a risk of the scheduler performing worse. Portfolio scheduling reduces this risk by introducing a portfolio of policies from which the scheduler can select one to be applied at each scheduling decision moment. It does so through simulating the performance of each policy if applied given the current queue of workload requests and workloads already active in the system. Based on a utility function, which can be things like minimizing operational risk or CPU ready time (the amount of time a CPU wanted to run but wasn't able to), the portfolio scheduler chooses a policy.

2.3.1 Stages of a Portfolio Scheduler

A portfolio scheduler goes through multiple stages Figure 2 depicts the four stages of a fully equipped portfolio scheduler as decribed by Deng et. al [8]. They are:

1. A portfolio of scheduling algorithms, which can be used to map a set of virtual machines to physical hosts, is *created*. The creation of the portfolio is currently done through manual selection of both common scheduling policies and policies specific for the utility function at hand.
2. A *selection* is made based on a, possibly changing, utility function.
3. The selected policy is *applied* to place VMs on physical hosts until the next selection moment.
4. The performance of the scheduler is *reflected* upon, which can possibly result in changes in the parameters or the portfolio of the scheduler.

2.3.2 Related Work on Portfolio Schedulers

Previous work on portfolio schedulers has provided meaningful solutions in various areas, once again confirming what we already know, that no individual policy performs better than all others for complex workloads [4][5]. For example, Van Beek et. al have implemented a portfolio scheduler with a utility

function aimed at reducing operational risk for datacenters running business-critical workloads [6]. Intel has used a version of a portfolio scheduler with a utility function aimed at reducing wait-time for jobs [9]. Ma et. al [7] proposed a portfolio scheduler for complex industrial workflows, which display periodic patterns, and extended it with the capability to reflect on its historic scheduling decisions. However, this addition to the portfolio scheduler, which uses reinforcement learning, does not adapt the actual portfolio of policies. The solution uses q-learning to refine the policy selection mechanism of the portfolio scheduler. Previous solutions therefore all lack the ability to reflect and adapt on the initially created portfolio.

3 Design of the Portfolio Scheduler and Genetic Algorithm

In this chapter we answer RQ1 by describing the design of an energy efficiency aware self-reflective portfolio scheduler for business-critical workloads. We first go over the functional and non-functional requirements in Section 3.2. Then we present the high level design in Section 3.3 and finally the detailed design decisions in Section 3.4.

3.1 Overview

We answer RQ1 by answering the sub-questions RQ1.1 and RQ1.2. The configuration of a portfolio scheduler has four stages: Equip the portfolio with policies, select which policy to apply, apply the selected policy and reflect on the constructed portfolio. Manual reflection on the performance of a portfolio and adapting it accordingly requires a lot of expertise and knowledge from human system administrators. By designing an automated reflection component for the portfolio scheduler and therefore answering RQ1.1, we can reduce complexity for system administrators. A proficient enough reflection stage could even make the initial selection of policies near trivial, as it will automatically adapt the portfolio to better configurations.

Furthermore, to reduce the massive carbon footprint of datacenters [17] and reduce operational costs, energy efficient scheduling is increasingly important. Answering RQ1.2 by designing a portfolio scheduler to optimize for energy efficiency could help create a more energy efficient scheduler.

We follow in this work the AtLarge Design Process [22] by first formulating the requirements of the portfolio scheduler (stage 1), after which we iteratively create the design by bootstrapping the creative process (stage 3) and constructing a high-level and low-level design (stage 4).

3.2 Requirements of the Portfolio Scheduler

To determine the requirements that should be addressed by the portfolio scheduler we first go over the stakeholders and use cases involved in a portfolio scheduler with a reflection component. We then formulate the functional and non-functional requirements of the system.

3.2.1 Stakeholders

We identify the following stakeholders of the system:

(S1) Customers: While customers do not interact directly with the system, they are influential for the demands and requirements that a cloud provider has regarding their datacenter performance. Customers expect high performance at a low cost and low downtime.

(S2) Datacenter Operators: Datacenter operators manage the operation of the datacenter infrastructure, this includes overseeing the behavior and performance of the scheduler.

(S3) Researchers: Portfolio scheduling is a relatively new scheduling technique. Before being adopted by datacenters, it is essential that the risks and benefits of new techniques are thoroughly researched.

3.2.2 Use Cases

Based on the identified stakeholders, we formulate four use cases where the system could be useful:

(UC1) Risk management: As shown in previous work [6], portolio scheduling can help manage the risk of sporadic bad performance of a single scheduler.

(UC2) Adapting the scheduler: This system could extend previous work in portfolio scheduling [6, 7, 8] with a reflection component that allows a datacenter operator to use the reflection component after a certain period of operation, to get suggested adaptations to the portfolio, which will likely improve it.

(UC3) Green efforts: The system can help improve their image towards customers, by showing that the cloud provider is putting effort in reducing their environmental impact.

(UC4) Research: Researchers can use past performance data of different policies to research the impact policy selection can have on various performance metrics.

3.2.3 Functional Requirements

The architectural requirements for the self-reflective energy aware portfolio scheduler are:

(FR1) Save system snapshots: The system should be able to save snapshots of the system, meaning the current queue of workloads and the active workloads mapped to hosts. Without this the system cannot rebuild system states to reflect on historic selection moments for the portfolio scheduler in the future.

(FR2) Load system snapshots: The system should be able load snapshots of the system, meaning the queue of workloads and the active workloads mapped to hosts for a historic system state. Without this the system cannot load the current system state in the portfolio scheduler simulator, or load historic system states to reflect on in the reflection stage.

(FR3) Provide a portfolio of policies: The system should enable the user to be able to provide an arbitrary set of scheduling policies to use for the portfolio scheduler.

(FR4) Reflect on the portfolio: The system should provide historic performance data on the portfolio scheduler and potentially adjust the portfolio accordingly.

3.2.4 Non-Functional Requirements

(NFR1) Facilitate reproducible experiments: Our experiments should be fully reproducible to strengthen the confidence in our results and allow for possible future comparisons with our results.

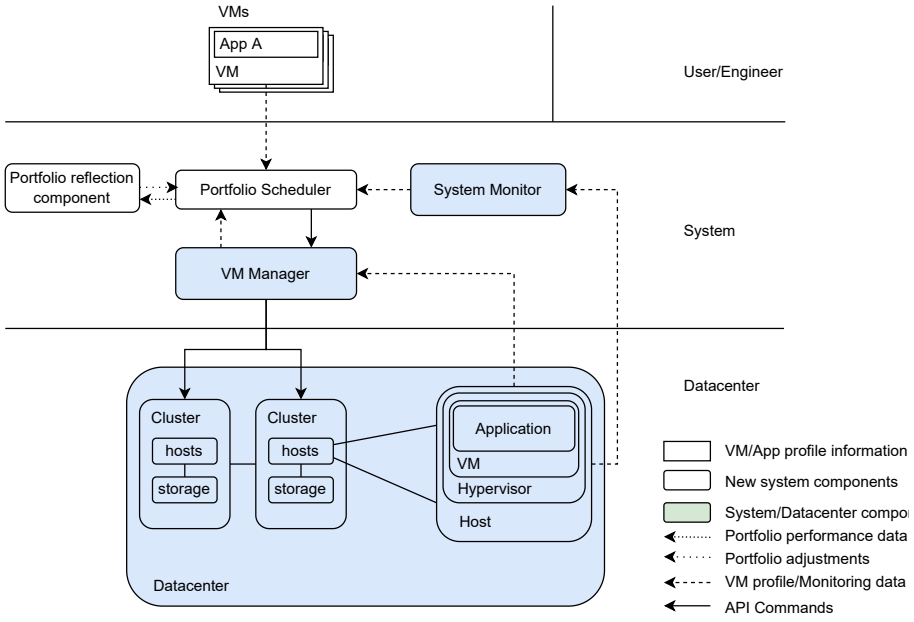

Figure 3: High level design of the portfolio scheduler, showing how the portfolio scheduler and the reflection component fit in the different layers of a datacenter.

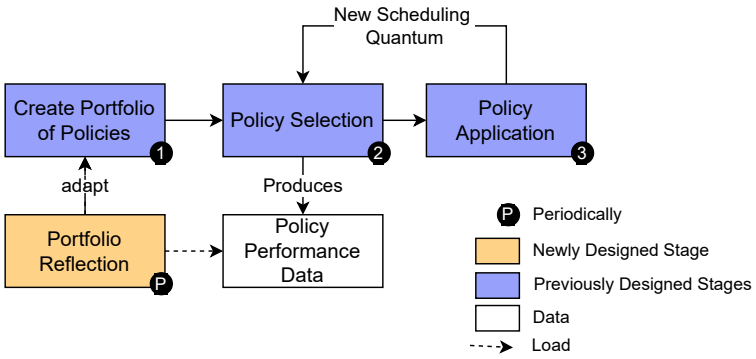

Figure 4: Decision cycle of the portfolio scheduler.

3.3 High-level Design of the Portfolio Scheduler

In this section we outline the high level design of the portfolio scheduler and how it fits into the different layers of a datacenter. We provide the high level design in Figure 3.

We extend previous work in portfolio scheduling by designing a component for the reflection stage that tries to find adjustments to the portfolio that are likely to improve the performance of the scheduler. Portfolio schedulers in previous work go through the first three stages: first a portfolio of policies is created, then for each scheduling moment in the data center, a policy is selected based on a certain utility function, and finally the policy is applied for the workload queue. Figure 4 depicts these three stages, and how the reflection stage designed in this work fits into this cycle. The reflection stage can periodically (P) be called to take historic performance data produced by the selection stage and based on this adapt the portfolio of policies. The portfolio scheduler fits in with the different system layers in the same way as in previous work, with the applications running on VMs in the user space, and the portfolio scheduler interacting with the system monitor and VM manager components in the System layer.

3.4 Detailed Design Decisions

In this section we describe some of the detailed design decisions made throughout the design process. First we describe which scheduling policies for VM provisioning are included in the base portfolio. Second, describe the simulator

used for policy selection. Finally we describe the design of the evolutionary algorithm used in the reflection stage of the portfolio scheduler.

Acronym	Policy Description
LCL	Lowest CPU Load: Selects the host with the lowest CPU load.
LML	Lowest Memory Load: Selects the host with the lowest memory load.
MCL	Maximum Consolidation Load: Selects the host with the largest gap between requested and actually used resources.
FF	First Fit: First Fit selects the first host on which a VM fits.
LCD	Lowest CPU Demand: Selects the host with the lowest CPU Demand.
VCPU	vCPU Capacity: Selects the host with largest the difference in required vCPU capacity and the available CPU capacity.

Table 1: Descriptions of the policies used for VM placement in the portfolio scheduler. These policies are by themselves not aware of energy-efficiency.

3.5 The Porfolio of Policies (FR3)

We initialize the portfolio scheduler with set of commonly used VM scheduling policies found in literature [28]. This will serve as a baseline for the portfolio scheduler. The scheduler used to create each policy is modeled after the Filter Scheduler architecture from OpenStack[3]. In this model the scheduler first *filters* hosts based on a set of policies configured by the user. This filtering can be based on things like available RAM, VCpu capacity, or host availability. After filtering the hosts, the remaining hosts are *weighed* based on an arbitrary number of policies and their respective weights. The host on which the VM will actually be scheduled on is then chosen from a subset of the highest ranking hosts. A description of the baseline set of policies can be seen in Table 1. By themselves, none of the included policies are aware of energy-efficiency.

[3]https://docs.openstack.org/nova/latest/admin/scheduling.html

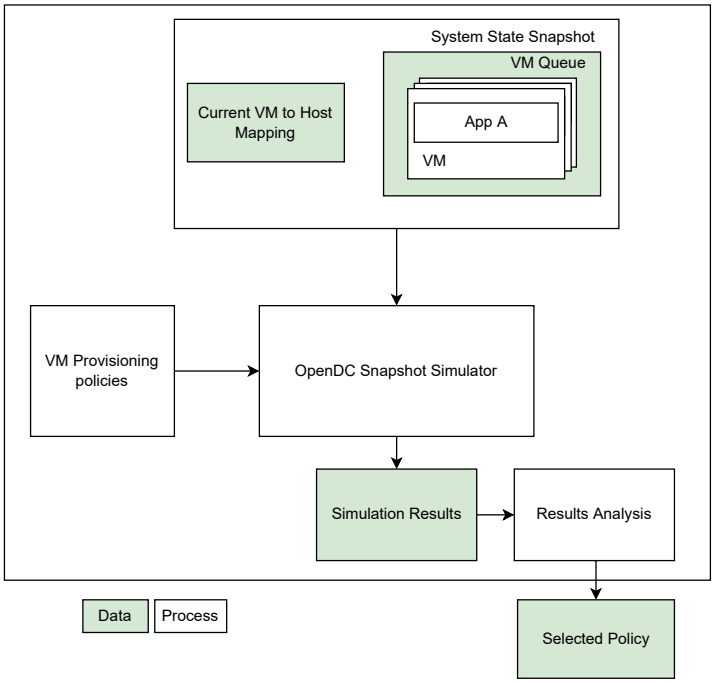

Figure 5: Overview of the portfolio scheduler simulator.

3.6 Energy Efficient Policy Selection

A portfolio scheduler selects a policy through simulation. Figure 5 depicts an overview of the simulator. The simulator takes as input a snapshot of the current system state, which consists of the queue of VMs waiting to be scheduled, the already active VMs in the datacenter, mapped to the hosts on which they are running and the workload profile of all VMs for a duration in the future. This is all that is required to reconstruct the system state. The simulator loads the snapshot, fullfilling FR2, then simulates each policy, for a set duration, and calculates the score per policy, based on a utility function. The best scoring policy is then applied by the scheduler until the next scheduling decision. Based on analysis of the policy selection in this work we see that decisions between schedulers are usually taken based on small differences with an average of 0.5% between best and worst performing policies. We do note that these small differences add up over time. However, similar to previous work in portfolio scheduling [6], we also see that there can be spikes of as much as 49% in the difference in performance of the best and worst policy. Portfolio scheduling can reduce the risk of a single policy suddenly performing worse.

Previous work has equipped portfolio schedulers with utility functions that select a policy based on risk of not meeting service level agreements (SLAs) [6] or on the balance between charged cost and job slowdown [8]. Because we are trying to improve the energy efficiency of the scheduler, an appropriate metric to use as the utility function is essential, as it determines which policy is selected.

Reddy et al. [24] explore metrics for sustainability for different components of a datacenter. These metrics range from describing the power efficiency of a whole facility, to that of a single server. Because in this work we are solely interested in energy efficient VM placement, and not in optimizing for datacenter topology or other components of a datacenter, we focus on the server level energy efficiency. The most commonly used metric to express this in is the 'performance per energy consumed' metric, e.g. 62.684 GFLOPS/watt for the Frontier TDS, the currently highest ranked supercomputer in terms of energy efficiency according to the Green500[4]. If we look at the *SPECpower_ssj2008* benchmark[5], which measures CPU power consumption at different levels of CPU utilization based on real-world data provided by hardware vendors, we see that the performance to power ratio of CPUs in the benchmark can vary from as low as 3,457 at 10% CPU utilization, to 12,124 at 80% CPU utilization, resulting in almost 4 times the workload operations done per unit of power. Considering CPU utilization levels below 15% are common in industry [13, 29, 30], the energy consumption of CPUs is all but efficient and there is a lot of improvement to be made regarding energy efficient VM placement. In OpenDC we can monitor the performance per energy as the amount of CPU cycles actually utilized in MHz per kJ for a host, since a host with no servers on it doesn't use any CPU cycles in OpenDC. We select this performance per energy consumed metric as the utility function used in the portfolio scheduler. The scheduler therefore selects the policy which during the simulation has the highest score for this utility function.

3.7 The Portfolio Reflection Component (FR4)

In order to reduce the complexity for human system administrators in maintaining a portfolio scheduler we design a reflection component for a portfolio scheduler capable of generating additions to the portfolio that are likely to be selected and improve the scheduler's performance, based on historic system state snapshots. With a reflection stage capable of suggesting improvements to the portfolio, the portfolio scheduler could become even more resistant to changes in workload characteristics, as changes in workload characteristics could mean the policies currently in the portfolio are no longer as efficient, and a new policy needs to be added. With this design we set an important step towards the design of a full generally capable portfolio scheduler by filling in the reflection component.

[4] https://www.top500.org/lists/green500/2022/06/

[5] https://www.spec.org/power_ssj2008/results/res2022q1/power_ssj200820211221-01146.html

To reflect on the performance of the portfolio scheduler we design an evolutionary algorithm [31] that tries to find a scheduler configuration that when added to the portfolio, is likely to be selected and improve the performance of the portfolio scheduler. Because a scheduler configuration can consider many different metrics while provisioning a VM, and many different parameter settings, the magnitude of the design space quickly becomes too large to consider all possible configurations. A scheduler configuration (an individual) in the design space is therefore encoded as a set of chromosomes, which can be altered through crossover or a set of mutators. Through these operators an evolutionary algorithm can explore large search spaces. The main challenge in designing an evolutionary algorithm comes from the non-trivial tasks of encoding an individual in the design space in a set of chromosomes that can explore the search space and from designing a fitness function capable of evaluating how well an individual performs for the problem at hand.

3.7.1 The Magnitude of the Design Space

To show the need for an evolutionary algorithm, we quantify the magnitude of the design space. For configuring a VM scheduler we define the magnitude of the design space as the size of the different number of (non-redundant) combinations of parts that together form a scheduler. We use the Filter Scheduler architecture as described in Section 3.5 with a static set of *filters* available in OpenDC that make sure the VM actually fits on the selected host. This means the magnitude of the design space is formed by the different combinations of weighers, the subset size and the vCPU over-provisioning parameter. We use a three-point estimation to investigate the size of the design space.

Best case: Suppose the weighers all have the same weight, and the subset size and vCPU overprosioning parameter are constant. Furthermore, the maximum number of weighers for a scheduler is set to 4. In this case, With the 8 different weighers available in OpenDC, the magnitude of the design space would be the number of different combinations of weighers to make up a scheduler. The formula to calculate the number of different combinations C with a sample size of r out of n different elements is $C(n,r) = \frac{n!}{r!(n-r)!}$. With a sample size of 1 *up to* 4 weighers and a total of 8 different elements, this would then be $\sum_{r=1}^{r=4} C(8,r) = 162$. With this magnitude a brute force search would definitely be a feasible approach.

Average case: Suppose a weight can have 20 different possible values, the maximum number of weighers is still 4, the subset size is a value between 1 and 32 and the vCPU Overcommit rate is a value between 1 and 48. The amount of different combinations would be as follows.
1 weigher: $C(8,1) \times 20 \times 32 \times 48 = 245,760$
2 weighers: $C(8,2) \times 20 \times 20 \times 32 \times 48 = 17,203,200$
3 weighers: $C(8,3) \times 20 \times 20 \times 20 \times 32 \times 48 = 688,128,000$
4 weighers: $C(8,4) \times 20 \times 20 \times 20 \times 20 \times 32 \times 48 = 17,203,200,000$
Total: $245,760+17,203,200+688,128,000+17,203,200,000 = 17,908,776,960$
Thus in this case we get a design space of over 17 billion points. An exhaustive

brute force search would be unfeasible.

Worst case: In reality, a datacenter could introduce even more configurable parameters and possible parameter values, so a brute force would still be infeasible.

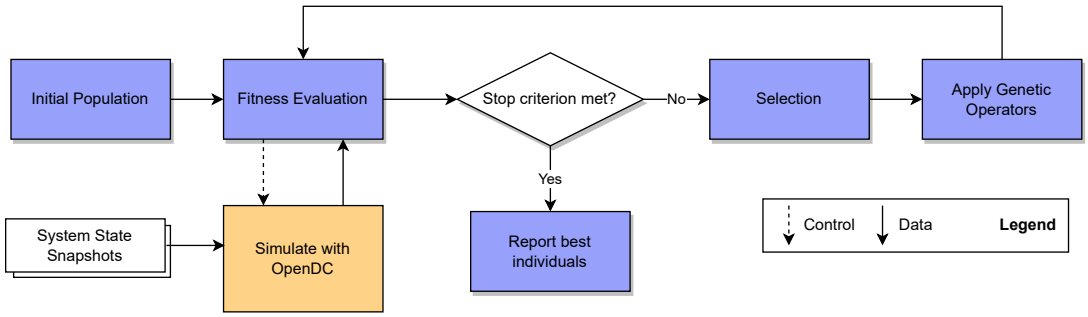

Figure 6: Exploration of scheduling policies using genetic search.

3.7.2 Genetic Optimization Process

The genetic optimization process for finding a scheduler configuration that improves the portfolio is shown in Figure 6. It goes through the following steps:

1. An initial population of random individuals is initialized based on the population size and seed provided.

2. The fitness of each individual is evaluated. This is done through simulation with OpenDC of the historic system state snapshots of each decision point of the portfolio scheduler. The fitness is then calculated as the sum of improved energy efficiency over all snapshots for the simulated duration, compared to that of the original portfolio of policies.

3. The stop criteria are evaluated. This can be things like a limit to the number of maximum generations, or convergence of the average population fitness. If the stop criteria is satisfied, the genetic optimization process ends and we jump to step 6 to report back the best individual.

4. If the stop criterion is not met, a selection of the fittest individuals is made from the population, which survive for the next generation.

5. Genetic operators like mutation and crossover are applied to the selection. The altering of individuals' chromosomes helps explore the design space. This new population is then evaluated once again in step 2.

6. Once the process completes the best individuals in the population are reported.

The concrete configuration of the genetic search algorithm is described in Section 4.2.2.

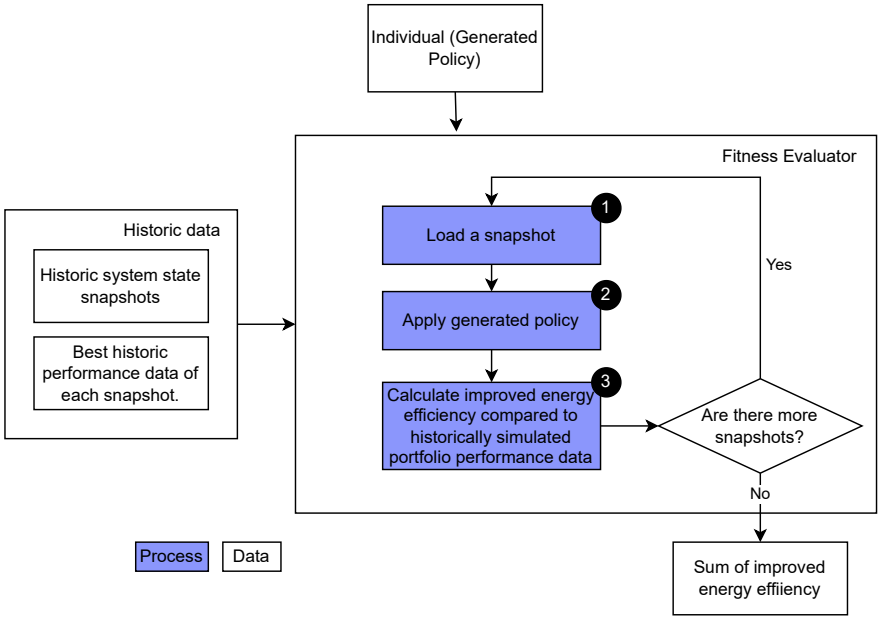

Figure 7: Design of the fitness evaluation for a generated scheduling policy using historic system state snapshots and performance of portfolio scheduler selection moments.

3.7.3 Fitness Evaluation

We design in this work an evaluator for new scheduling policies using historic data of selection moments of the portfolio scheduler. The design of the fitness evaluation is shown in Figure 7.

Designing a fitness evaluation capable of quantifying the performance of an individual is essential for any evolutionary algorithm. In our case the goal of the fitness evaluation is to steer the evolutionary algorithm to finding a scheduling policy that when added to the portfolio of policies improves the portfolio and is therefore chosen. The fitness evaluation needs to consider past selection decisions determine how much an individual improves on them. Because of this we

define the fitness function as the sum of improvement of the energy efficiency over all historic system snapshots of previous selection moments of the portfolio scheduler, compared to the original scheduler's performance. Because this fitness function uses historic snapshots and the best simulated performance of the original porfolio scheduler, it makes sure a policy is likely to be selected in the future and tries to optimize for the greatest improvement in energy efficiency for the simulated duration compared to the historically simulated results of the original portfolio scheduler.

Previous work only considers performance however, these considerations cannot address the need for energy-efficient decisions.

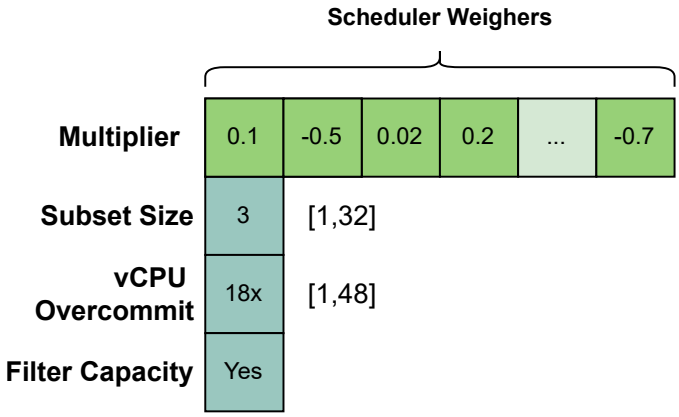

Figure 8: Genotype encoding of a scheduler configuration.

3.7.4 Encoding of Schedulers

We model the encoding of schedulers in the search space after the encoding of schedulers in the risk optimization genetic search algorithm as implemented by Radice [23]. This encoding can be seen in Figure 8. The scheduler used is modeled after the Filter Scheduler architecture from OpenStack[6] which is described in Section 3.5.

By combining different sets of filters and weighers, many different scheduling policies can be recreated. After all, when dissecting a datacenter scheduler in different components as done in [32], VM placement comes down to first filtering machines and then sorting based on certain criteria. We also include the size of the subset of best ranking hosts and the vCPU overcommit ratio in the genotype, to see their effect on the schedulers' performance.

[6]https://docs.openstack.org/nova/latest/admin/scheduling.html

3.8 Design for Reproducibility (NFR1)

We conduct all experiments in this work through OpenDC, which with discrete-event simulation and seeded randomness is deterministic in its simulation process. Similarly, Jenetics [33] allows for seeded randomness in the evolutionary algorithm, resulting in a deterministic process. Furthermore, to allow for reproduction of the experiments, the extensions to OpenDC done in this work as well as the experiments and all necessary data are publicly available on Github[7].

[7] https://github.com/Timovanmilligen/opendc

4 Evaluation of the Portfolio Scheduler

In this chapter, we answer RQ2 by designing and conducting an evaluation of the portfolio scheduler. We describe the implementation of a prototype according to the design in Section 3 and analyze it using different methods.

4.1 Overview

This chapter is structured as follows:

1. We describe the working prototype implemented in OpenDC and its details in Section 4.2.
2. The details of the setup used in the experiments is described in Section 4.3.
3. The metrics we use to evaluate the implemented prototype and how they are gathered is described in Section 4.4.
4. The performance of the baseline energy efficiency focused portfolio scheduler is evaluated in Section 4.5.
5. The ability of the added reflection stage to adapt and improve the portfolio is evaluated in Section 4.6.
6. The impact of the duration of the portfolio scheduler simulator on the ability of the portfolio scheduler to choose energy efficient policies is evaluated in Section 4.7.
7. In Section 4.8 we evaluate the impact of a different workload on the ability for the portfolio scheduler to choose energy efficient policies as well as on the performance of the reflection stage.

We summarize our contributions and the threats to the validity of our results in Section 4.9.

4.2 Implementation of a Prototype

We answer RQ2.1 by extending OpenDC [21] with a prototype built after the design as formulated in Section 3. We add to the simulator the capabilities to take and load system state snapshots. This capability to load historic system states can be useful for future extensions of the program or for research.

Using this added functionality we add portfolio scheduling functionalities to OpenDC. Aside from this we implement an evolutionary algorithm that takes a number of snapshots and searches for a scheduler configuration that tries to optimize for a given metric over all snapshots for a certain simulated duration.

Filter	Description
Host Availability	Filters on active hosts.
Available vCPUs	Filters hosts based on the vCPU requirements of a server and the available vCPUs on the host, taking into account the vCPU overprovisioning ratio.
Available RAM	Filters hosts based on the memory requirements of a server and the RAM available on the host.

Table 2: Static filters used throughout the experiments.

Weigher	Ranking Property
VM Count	The number of instances on a host.
Available RAM	Available host memory.
Available RAM (per pCPU)	Available memory divided by the number of physical CPUs
Available vCPUs	Available vCPUs considering the CPU overcommitment ratio.
vCPU Capacity	The difference in available CPU capacity and required CPU capacity.
CPU Demand*	CPU Demand.
CPU Load*	Current CPU utilization.
Maximum Consolidation Load*	The gap between requested and actually used CPU resources.

Table 3: Weighers available in OpenDC. **Added to OpenDC in this work.*

4.2.1 The Portfolio Scheduler

The individual policies used for placing VMs onto physical machines for all experiments are modeled after the filter scheduler as described in Section 3.5. The static set of filters that we use to make sure VMs actually fit onto the mapped hosts are listed in Table 2. We use a vCPU overprovisioning ratio of 16.0. The host weighers available in OpenDC and the weighers added to OpenDC in this work are described in Table 3.

The baseline portfolio scheduler is equipped with the commonly used policies [28] listed in Table 1. The portfolio scheduler simulates how each policy performs during a user configured duration in the future. In order to realize the design of the simulator in Figure 5, we extend OpenDC with the capability to create and load a snapshot of the current system state. In this way different policies can be simulated by loading a single snapshot and replaying it multiple times with a different scheduler.

Parameter	Value
Population Size	30
Stop Criteria	10 steady generations or a maximum of 50 generations.
Selection	Tournament Selection [34]
Genetic Operators	Uniform crossover (probability 0.2), uniform mutation (probability 0.15), Gaussian mutation (probability 0.10), length mutation (probability 0.02)

Table 4: Configuration of the genetic search algorithm.

4.2.2 Genetic Algorithm

We implement a genetic search algorithm according to the design in Section 3.7. We implement the genetic algorithm, which is modeled after the one used in Radice [23], using Jenetics [33], a Java Library for Genetic Algorithms. All possible weighers used in the search process are listed in Table 3. We will go over some of the parameter settings used, which are described in Table 4.

In this work we limit the maximum number of generations to 50, unless the average fitness of the last 10 generations differs at most 0.01% from the average fitness of the last 30 generations, in which case the fitness is considered converged.

To explore the design space we make use of four different genetic operators: *Uniform crossover*, *guassian mutation*, *uniform mutation* and *length mutation*. Uniform crossover swaps the genes at a random index in two parents' chromosomes to create new offspring with probability 0.2. Guassian mutation changes the parameter value of a gene to a random value based on a Guassian distribution around the current value, and is applied with a probability of 0.1. This allows for exploration of parameter values around the current values. Uniform mutation is used with a probability of 0.15, which changes the parameter value of a gene to a random value, to try and maintain more diversity in the population. Finally length mutation is used with a probability of 0.02, which in turn has a probability of $\frac{1}{3}$ of removing a weigher gene if possible, and a probability of $\frac{2}{3}$ of adding a weigher gene. For the selection stage we make use of tournament selection [34], a widely used selection mechanism due to its lack of stochastic noise and independence to scaling of the fitness function [35].

4.3 Experimental Setup

In this section we present our experimental design. Table 5 summarizes the design and setup for the different experiments.

ID	Experiment Focus (Section)	Workload	Reflection	Topology	Interference	Power Model	Simulator Duration
E1	**Baseline** Portfolio Scheduler (§4.5)	Solvinity	No	Solvinity Topology	Yes	Based on SPECpower_ssj2008 benchmark	20m
E2	Reflection Component (§4.6)	**Solvinity**	**Yes**	Solvinity Topology	Yes	Based on SPECpower_ssj2008 benchmark	20m
E3	PS Simulator Duration(§4.7)	Solvinity	No	Solvinity Topology	Yes	Based on SPECpower_ssj2008 benchmark	**40m and 60m**
E4	Workload(§4.8)	**Bitbrains**	**Yes**	Solvinity Topology	**No**	Based on SPECpower_ssj2008 benchmark	20m

Table 5: The design and setup of our experiments. In bold are the distinguishing features of each experiment.

4.3.1 Workload

We use in this work a long-term real-world workload trace obtained from Solvinity [36], a private cloud provider in the Netherlands, operating mainly in the Dutch ICT market. We use this trace as a baseline, as it is representative of real-world Dutch business-critical workloads. The trace, of which the characteristics can be seen in Table 6, spans a period of 3 months and 1,800 VM submissions running business-critical workloads. Furthermore, to evaluate the impact of workload on the performance of the portfolio scheduler and the reflection component, we use a second business-critical workload trace from the Grid Workloads Archive [37], namely Bitbrains [38]. Bitbrains is another real-life workload trace gathered by Solvinity. It spans a period of 1 month and consists of 1,250 VMs.

4.3.2 Datacenter Topology

We use for the experiments the original datacenter topology that was used to run the Solvinity workload trace. It consists of 12 computer clusters with a total of 162 physical hosts. The original topology is spread over three fiber-optic connected datacenters. The topology consists of 9 *standard* clusters and 3 *bigmem* clusters. The *standard* clusters contain 16 physical hosts, each equipped with 128GB of memory and 8-core CPUs. The *bigmem* clusters contain 6 physical hosts, each with 768GB of memory.

4.3.3 Power Model

We use in our experiments a power model available in OpenDC based on the *SPECpower_ssj2008* benchmark[8]. This benchmark measures power consumption at different levels of CPU utilization, and has been validated by SPEC.

4.3.4 Interference Model

Because performance interference is a common operational phenomena in datacenters [39], we use make use of a performance interference model to more realistically simulate this phenomenon.

[8]https://www.spec.org/power_ssj2008/results/res2022q1/power_ssj200820211221-01146.html

Workload	VM submissions/h		VM duration [days]		CPU Usage [MHz]		Memory Usage [GB]	
	Mean	σ	Mean	σ	Mean	σ	Mean	σ
Solvinity	0.919	24.286	44.702	21.171	542.068	2,229.732	7.552	10.840
Bitbrains	1.857	44.677	28.703	5.164	1,811.749	5,080.531	11.755	32.602

Table 6: Characteristics of the workloads used in this thesis

We use in our experiments a common model for workload interference [36], where a set of VMs is contending for the same CPU resources. The model is assigned a score from 0 to 1, where 1 indicates no CPU interference at a given CPU load, and 0 indicates full interference between VMs. The model is created from the placement data of the Solvinity workload, which ran on the Solvinity topology. It takes this placement data and the performance of a VM as the fraction of time which there was CPU interference subtracted from 1. Then when any originally co-allocated workloads from the times of interference are co-allocated during simulation, the CPU demand of a VM is set to this performance level with a probability of $\frac{1}{N}$, where N is the number of co-allocated workloads from the original placement data thus simulating interference. As this model was created from the Solvinity workload, it is only applicable for that workload.

4.3.5 Simulation Duration

The duration for which the portfolio scheduler simulates impacts the future system performance prediction capabilities of the portfolio scheduler. Since the portfolio scheduler simulator as implemented in this work uses a part of the future workload trace in its simulation, it is 100% accurate even for an infinite duration, which is unrealistic for any real-world implementation of a performance predictor. Previous work shows that performance predictors like, i.e. a cpu contention predictor can be accurate for around 20 minutes in the future [36]. For this reason we set the baseline simulation duration to 20 minutes.

We vary in the experiments in Section 4.7 the duration for which the portfolio scheduler simulator simulates the policies to evaluate the possible impact of the length of accurate performance prediction on the performance of the portfolio scheduler.

4.3.6 Snapshots used for Reflection

Because we want to know if the reflection component can use historic data to adapt the portfolio for future scheduling decisions, we use half of the system state snapshots produced by the portfolio scheduler for the genetic search process. This means 92 snapshots (up until a time of 55,405 minutes in the trace) for the Solvinity workload and 43 snapshots (up until a time of 22,840 minutes in the trace) for the bitbrains workload. The resulting policy from the genetic search is applied only after these points in the trace when evaluating the adapted portfolio. This point is identified by the vertical dashed red line at the mentioned timestamps.

4.4 Metrics Used for Evaluation of the Portfolio Scheduler

To evaluate the performance of the portfolio scheduler, we look at some common system metrics on both the level of physical hosts and the server level, like resource utilization and overcommitted CPU cycles, but also some portfolio scheduling specific metrics like policy distribution. We evaluate the result of the reflection stage in a similar fashion, but also look at genetic exploration related metrics like average population fitness to evaluate the exploration process.

The metrics used in this work are exposed by the telemetery system in OpenDC and sampled every 5 simulated minutes. We list the metrics relevant to this work in Table 7.

Name	Unit	Description
$scheduler.servers[state = pending]$	-	Number of VMs pending to be scheduled by the scheduler.
$scheduler.servers[state = active]$	-	Number of VMs currently active in the system.
$system.cpu.limit$	MHz	Total CPU capacity available to a host or VM.
$system.cpu.demand$	MHz	CPU capacity of the host requested to be utilized.
$system.cpu.usage$	MHz	CPU capacity of the host actually utilized.
$system.cpu.utilization$	%	CPU utilization relative to the capacity of a single CPU.
$system.cpu.time[state = active]$	s	Accumulated CPU time spent in a running state.
$system.cpu.time[state = idle]$	s	Accumulated CPU time spent in an idle state.
$system.cpu.time[state = steal]$	s	Accumulated CPU time requested by a VM, but not provided due to CPU contention.
$system.cpu.time[state = lost]$	s	Accumulated CPU time requested by a VM, but not provided due to hypervisor overhead.
$system.power.usage$	W	Active power usage of the host.
$system.power.total$	J	Accumulated energy usage of the host.

Table 7: Metrics exposed by OpenDC relevant to this work.

4.5 Performance of the Baseline Portfolio Scheduler

Our main findings for this experiment are:

MF1 The baseline portfolio scheduler can not outperform all individual policies in terms of energy efficiency.

MF2 There is significant difference in CPU steal and lost time between the different policies.

To analyze the performance of the baseline portfolio scheduler we first look at how well it can optimize for energy efficiency, and how this holds up to the individual commonly used policies [28] included in the portfolio scheduler, as described in Section 3.5. Then we look at the distribution of selected policies to see if each policy is useful (selected). We use the setup described in Section 4.3.

Figure 9 shows the overall energy efficiency of the portfolio scheduler compared to the individual policies included in the portfolio for the Solvinity trace. We see that while it is able to outperform almost all of the individual policies, the *lowest memory load* policy still outperforms the portfolio scheduler by 0.31% with an overall power efficiency at the end of the trace of 150.85, compared to the 150.39 of the portfolio scheduler (MF1). This could be an indication that the simulated duration from the point of policy selection is not indicative enough of future performance of a VM for this workload for the portfolio scheduler to make the right decision. It could also mean that the LML policy is already well optimized for energy efficiency. We further explore these ideas in Sections 4.7 and 4.8.

Figures 11 and 12 give more insight in how the power efficiency of each scheduler is made up. The figures depict the total power draw and total CPU usage (work done) for each scheduler. We see that the power draw for the LML scheduler and the portfolio scheduler do not differ much, but the LML scheduler gets more work done for the same amount. This could be due to multiple factors, like the CPU steal time (also called CPU ready time), which is the amount CPU time a VM is ready to run, but was not able to, due to the CPU lost time, which is the amount of CPU time lost due to workload interference, and because of more energy efficient placement in terms of CPU utilization.

The average and standard deviation of server CPU steal time and CPU lost time are listed in Table 8. We see that the *lowest cpu demand* policy has by far the lowest combined steal and lost time, which makes sense since it spreads out the VMs based on CPU demand, resulting in less consolidation of virtual machines on physical machines. The best performing policy in terms of energy efficiency (LML) has on average half the CPU steal time compared to the portfolio scheduler, which could explain the difference in total CPU usage. Most of the policies experience significant CPU contention. With an average VM duration of 44.7 days for the solvinity trace, VMs experience CPU contention 0.26% of the time with the portfolio scheduler, and lose 0.65% of time due to

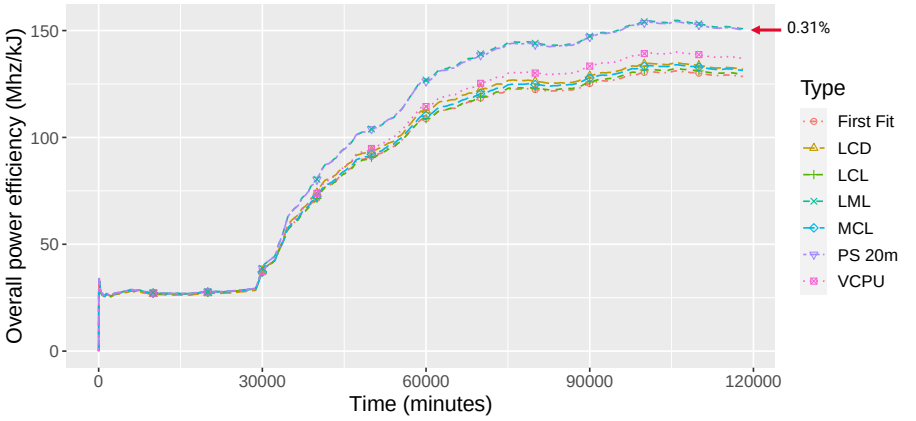

Figure 9: Overall power efficiency of the individual schedulers included in the baseline portfolio scheduler and the portfolio scheduler itself for the Solvinity trace.

interference between VMs. With the standard deviations. Because it is common to refund customers that experience on average a CPU interference above 2.5% on the day according to refund policies of cloud providers like Amazon Web Services[9], Microsoft Azure[10] and Google Cloud[11], we can clearly see these SLAs would have been broken for a significant portion of the VMs, considering the large standard deviation. We find that policies that are not already well optimized for reducing steal and lost time, or take it into account specifically, experience significant performance degradation due to these operational phenomena (MF2). On closer inspection we observe that 10.9% of VMs experience lost CPU time greater than 2.5% of their active CPU time for the portfolio scheduler. The *lowest cpu demand* policy performs the best with 3.78% of VMs above this threshold.

Regarding the selected policies, we see in Figures 10 and 13 that only 4 out of the 6 included policies are ever selected, thus meaning that 2 out of 6 policies are not useful for the current portfolio and workload. Interestingly, the LCL policy is selected by far the most, while being near the bottom in terms of energy efficiency as an individual policy.

[9] https://aws.amazon.com/compute/sla/
[10] https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_9/
[11] https://cloud.google.com/compute/sla

Scheduler	Steal Time (minutes)		Lost Time (minutes)	
	Mean	σ	Mean	σ
FF	517.8	2,955.2	445.7	3,432.4
LML	82.2	616.9	455.6	3,530.0
LCL	593.1	3,438.7	389.3	2,841.1
LCD	71.5	644.2	149.0	2,310.7
MCL	488.7	2,867.3	467.4	3,717.0
VCPU	454.9	2,862.9	523.2	3,838.4
PS	167.0	1,284.6	419.3	2,864.2

Table 8: Average server CPU steal time and CPU lost time of the different schedulers for the Solvinity trace.

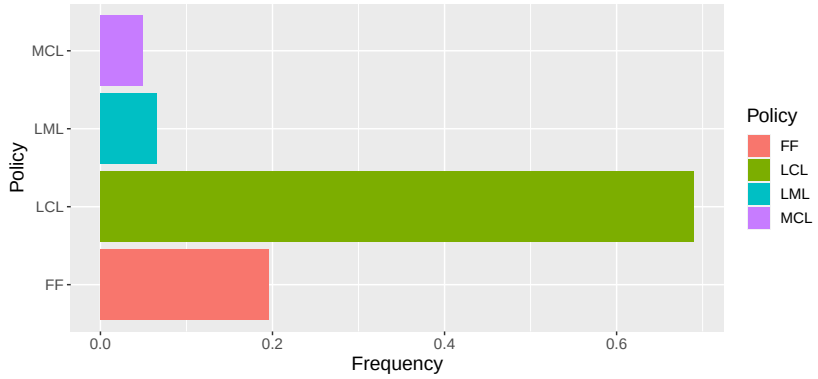

Figure 10: Distribution of selected policies for the baseline portfolio scheduler.

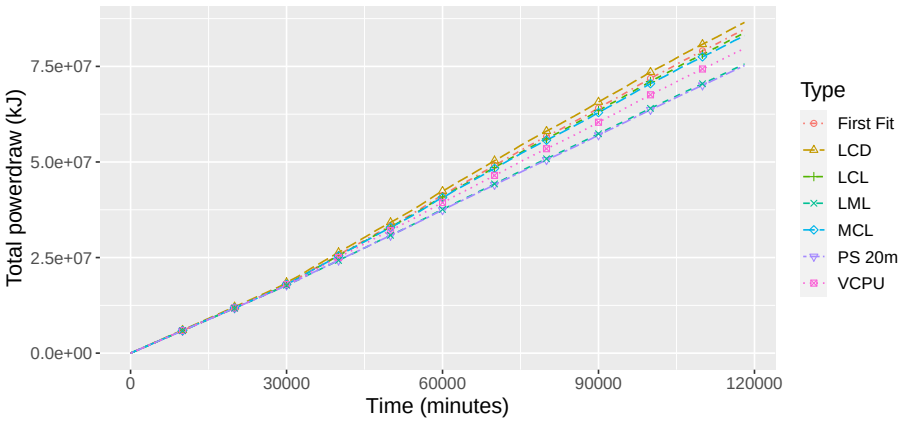

Figure 11: Total powerdraw of the individual schedulers included in the baseline portfolio scheduler and the portfolio scheduler itself for the Solvinity workload.

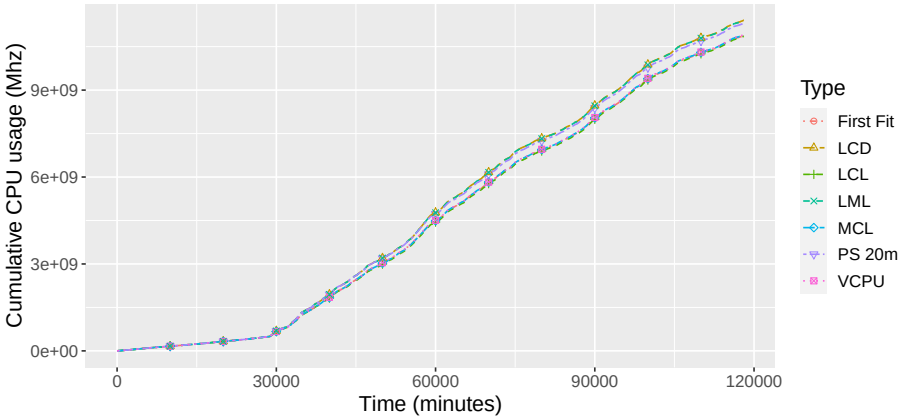

Figure 12: Cumulative CPU usage of the individual schedulers included in the baseline portfolio scheduler and the portfolio scheduler itself.

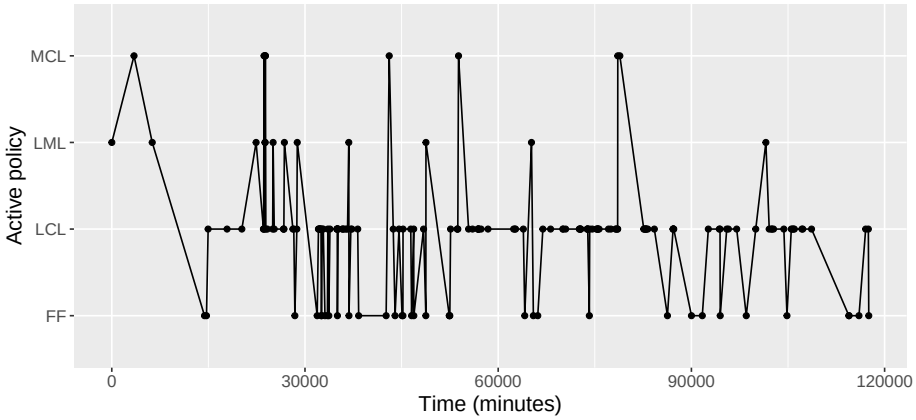

Figure 13: The active policy of the baseline portfolio scheduler over time.

4.6 Performance of the Reflection Component

Our main findings for this experiment are:

MF3 The reflection component can produce policies that improve on past selection moments.

MF4 The adapted portfolio does not significantly improve the energy efficiency for future scheduling decisions for the baseline workload, even though the added policy is often selected.

MF5 The level of improvement over historic selection moments is not entirely representative for improvements in future selection moments.

The solvinity workload resulted in 184 selection moments for the portfolio scheduler and thus in 184 system state snapshots. Through genetic exploration the reflection component tries to find a scheduler configuration that improves the energy efficiency the most over the first 92 of these snapshots for the simulated duration. We see the progress of the elite fitness value and the best policy configuration in Figure 16. We see that the genetic exploration immediately finds a small improvement over the current portfolio based on the simulation, and a steep increase in fitness in the first 8 generations (MF3). It then continues to switch between the different weighers and parameters for small improvements until it converges. The performance of the portfolio after adding the found scheduler configuration can be seen in Figure 14. We would expect the adapted portfolio to perform better, but we see that it performs within 0.01% of the original portfolio (MF4), even though the reflected policy is chosen a significant amount of the selection stage, which is depicted in Figure 15. The simulated duration at portfolio scheduler selection moments for the Solvinity trace could be unrepresentative of future performance when it comes to energy efficiency based on these results, as improvements in the short term of 20 minutes, can lead to worse energy efficiency in the future.

On closer inspection of pending VM requests depicted in Figure 17, we see another possible reason for the minuscule difference in performance, and the lacking performance of the reflection component. We see that out of the 1800 VM workloads in the Solvinity trace, around 1,000 of them arrive in a short burst, with a few similar bursts of a few hundred VMs. This biggest spike in VM requests happens before our reflection moment, meaning this single scheduling moment could have the biggest impact on the genetic search process and final overall power efficiency. If we run the elite scheduling policy from the genetic search of the reflection stage on the snapshots related to the three spikes in VM requests, at 0, 28800 and 43080 minutes in the trace, we see that it does not score higher than the orginally selected schedulers for all of these moments. This means the reflection stage was not dominated by these spikes.

Still, the generated policy is often chosen, yet does not improve the energy

efficiency. When analyzing the difference in energy-efficiency between the generated policy when it is selected and the second best policy, which is a total of 30 times, the average difference is 0.0099. These very small improvements lead to the miniscule improvement in energy efficiency for the extended portfolio scheduler. They also mean that the improvement seen in the historic selection moments is significantly higher than in future selection moments, meaning the level of historic performance increase is not entirely representative for future improvements (MF5).

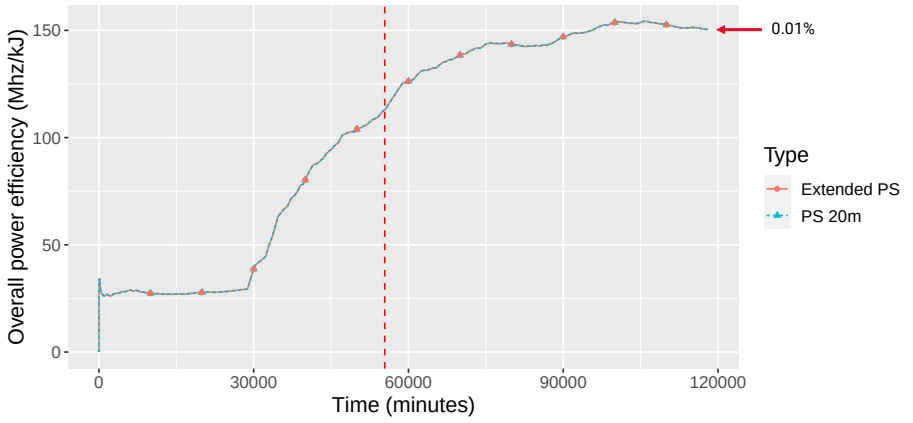

Figure 14: Overall power efficiency of the baseline portfolio scheduler before and after an iteration of the reflection stage for the Solvinity trace. The vertical red line shows the moment of reflection and portfolio adaption.

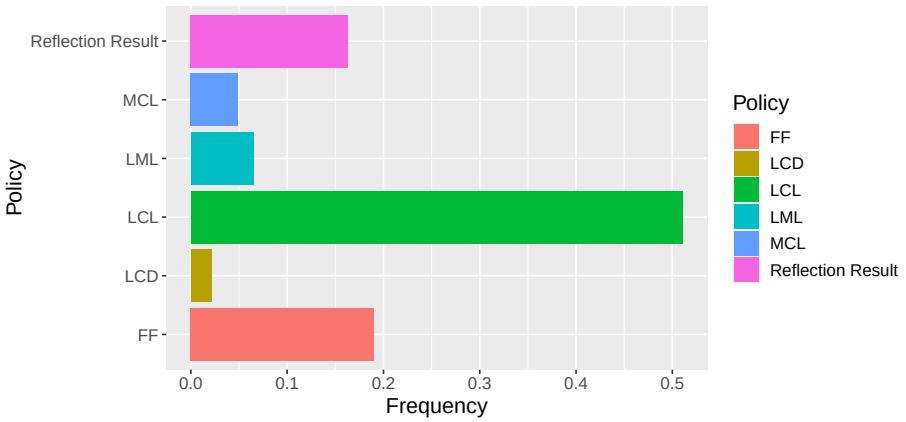

Figure 15: Distribution of selected policies for the adapted portfolio scheduler for the Solvinity trace.

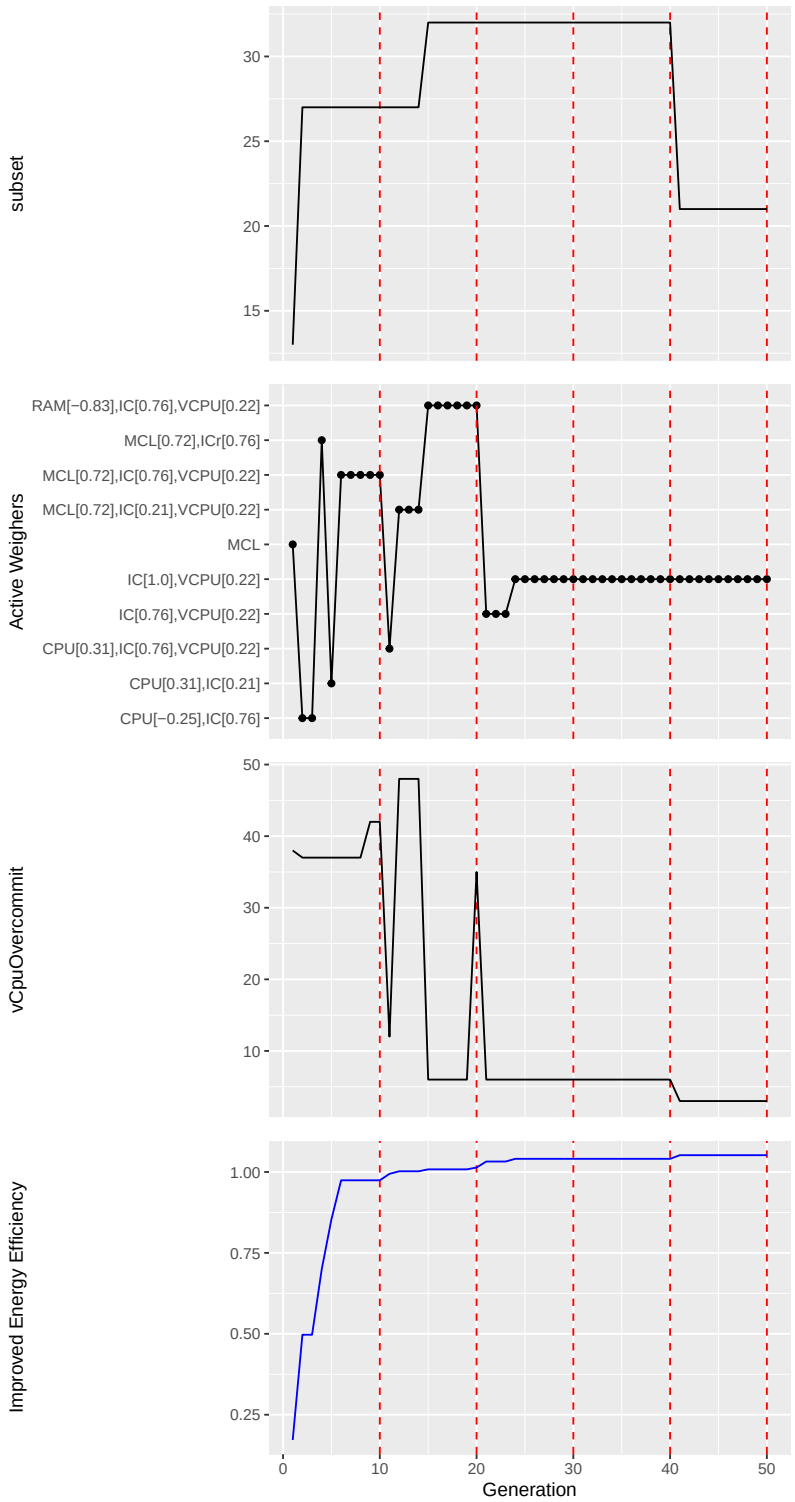

Figure 16: Elite fitness and scheduler configuration for the genetic exploration per generation for the historic system state snapshots of the Solvinity baseline experiment. **MCL = MaximumConsolidationLoad Weigher, RAM = Ram Weigher, CPU = CPU Load Weigher,VCPU = VCPU Capacity Weigher,IC = Instance Count Weigher.*

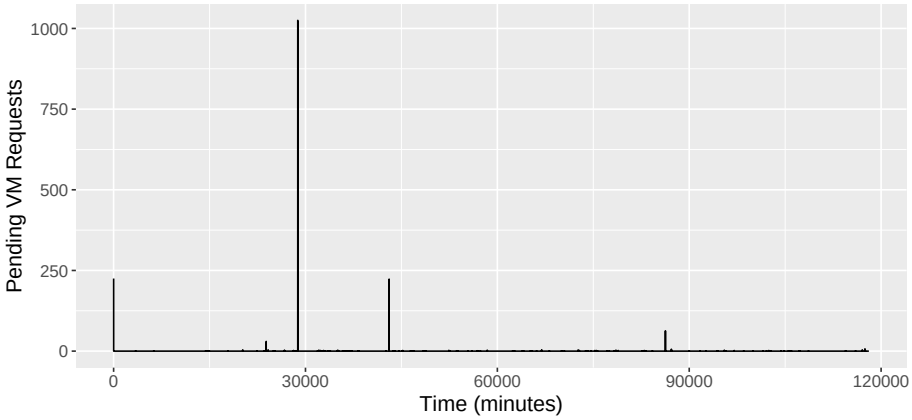

Figure 17: Pending VM requests over time for the Solvinity trace.

4.7 Impact of Portfolio Simulation Duration on the Performance of the Portfolio Scheduler

Our main findings for this experiment are:

MF7 Longer simulation accuracy influences which policies are selected.

MF8 Accurately predicting a policy's performance for a longer duration does not necessarily result in better policy selection decisions in the long run.

To evaluate the impact of the simulator duration on the performance of the portfolio scheduler, we experiment with a simulated duration of 40 and 60 minutes. For a longer duration the performance during the simulation could be more representative of future performance, increasing the performance of the portfolio scheduler.

We see however in Figure 18 that this is not necessarily the case. For a duration of 40 minutes the portfolio scheduler improves upon the overall energy efficiency of the baseline of 20 minutes at the end of the trace by 0.2%, indicating it can make at least some better scheduling decisions due to the simulator's increased performance prediction accuracy. Figure 19 depicts the new frequency of selected policies for a simulator duration of 40 minutes. We see that the change comes from the *first fit* policy being selected less and making room for the other policies, as well as *lowest cpu demand* being selected (although only once) (MF7).

For a simulated duration of 1 hour, the overall power efficiency performs significantly worse than the other durations. Upon inspection of the selected policies we see that this is because it selects a different policy at the peek of

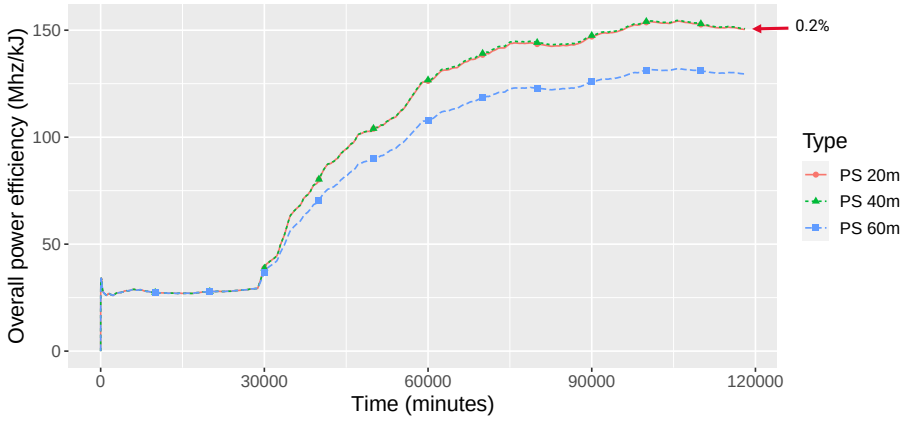

Figure 18: Overall power efficiency of the portfolio scheduler for different simulation durations for the Solvinity trace.

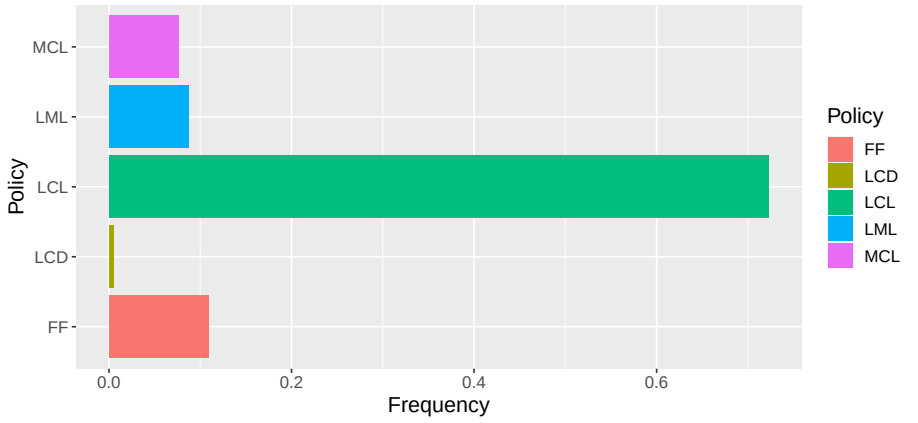

Figure 19: Distribution of selected policies for the portfolio scheduler using a simulator duration of 40 minutes.

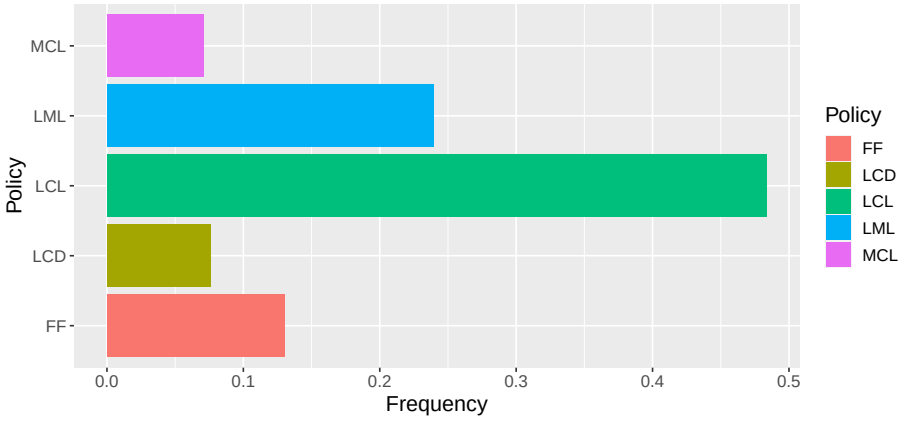

Figure 20: Distribution of selected policies for the portfolio scheduler using a simulator duration of 60 minutes.

VM scheduling requests depicted in Figure 17. This means a larger simulation duration does not necessarily always mean better policy selections in the long run (MF8). Figure 20 shows the frequency of policies chosen when using a simulation duration of 60 minutes. We see that the *lowest memory load* policy has increased significantly in the fraction of times it is selected, as well as the *lowest cpu demand* policy. We therefore find that increased performance prediction accuracy has a significant influence on the selection of policies in a portfolio scheduler. Migrating virtual machines after the selected host to schedule the VM on starts performing worse regarding energy-efficiency seems essential for long-running virtual machines.

4.8 Impact of Workload on the Performance of the Portfolio Scheduler

MF9 Workload characteristics can have an impact on the energy efficiency optimizing capabilities of the portfolio scheduler.

MF10 The performance per unit of energy metric does not consider trade-offs that could be well worth it for cloud service providers.

In this section we evaluate the impact of the workload on the performance of the portfolio scheduler and on the implemented reflection component. We choose another real life business-critical workload trace also gathered by Solvinity, called Bitbrains. The workload characteristics are described in Table 6. The experimental setup is the same, except for the interference model which is specific to the baseline workload and inapplicable to this workload. Figure 21

depicts the power efficiency of the baseline portfolio scheduler and the individual policies for the Bitbrains workload trace. We see at the end of the simulation,

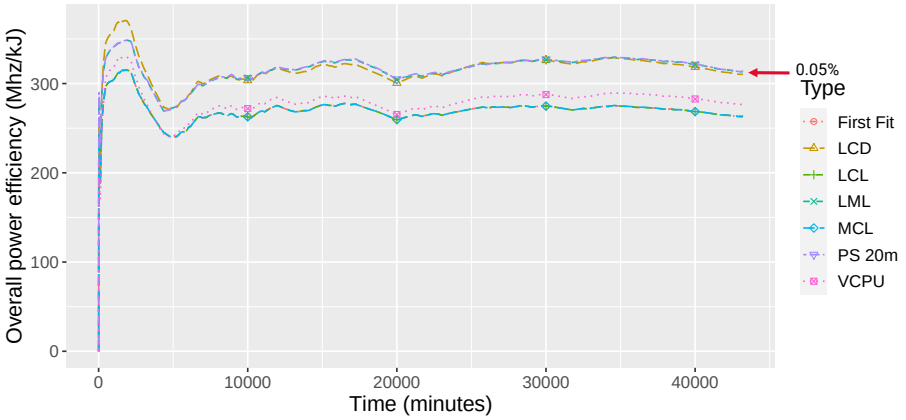

Figure 21: Overall power efficiency of the individual policies and the portfolio scheduler for the Bitbrains workload.

that the portfolio scheduler outperforms all individual policies in terms of energy efficiency (MF9). It outperforms the second best policy (LML) by 0.05% in terms of overall power efficiency and the second best policy (LCD) by 1.05% which is a more significant improvement when it comes to power efficiency in the long term.

To get more insight in how the energy efficiency metric of the individual policies is made up, we look at total powerdraw in Figure 22 and the CPU work done in Figure 23. We see that the third highest scoring policy, *lowest cpu demand*, while having done 24.82% more CPU work, has done so at an increase in total power draw of 26.12% compared to the portfolio scheduler, resulting in the worse power efficiency of 1.05%. This was not obvious from just looking at the energy efficiency metric (MF10). The extra work done is very significant however, and could well be worth the trade-off. Table 9 depicts the CPU steal times of the different schedulers. We see that the *lowest cpu demand* policy outperforms all other policies in terms of CPU steal time significantly as well, even though it is only the second best in terms of energy efficiency. A more balanced utility function that takes other performance metrics aside from energy efficiency into account could help improve the overall performance of the portfolio scheduler.

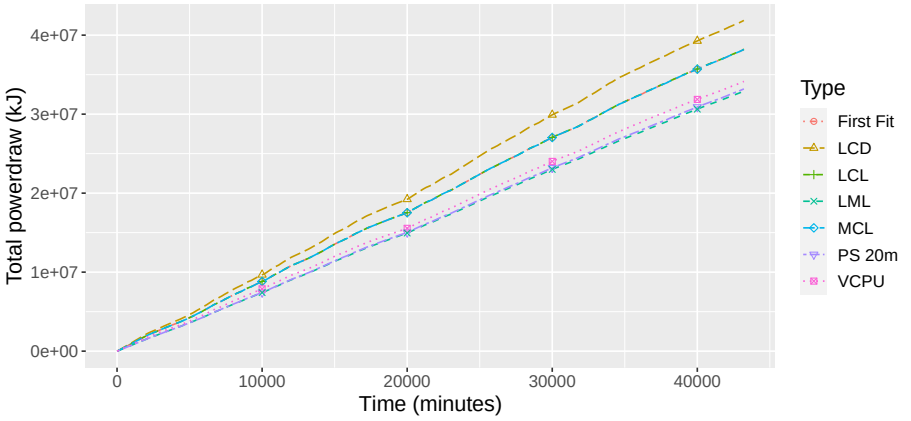

Figure 22: Overall power draw of the individual policies and the portfolio scheduler for the Bitbrains workload.

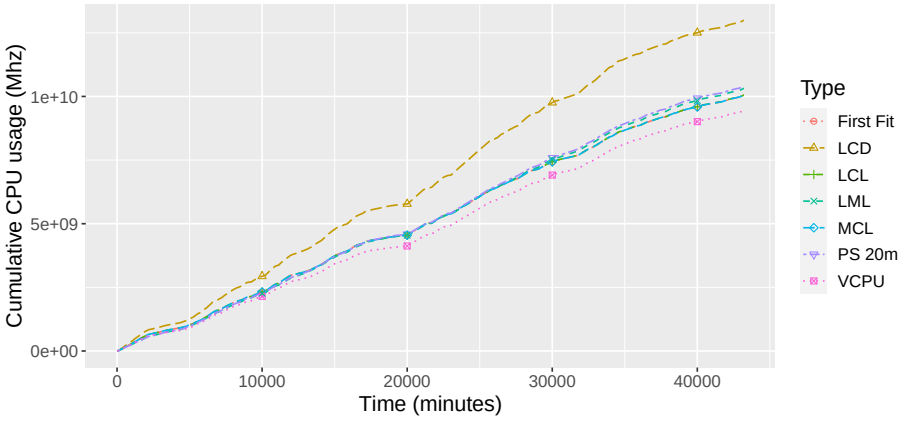

Figure 23: Overall CPU usage of the individual policies and the portfolio scheduler for the Bitbrains workload.

Scheduler	Steal Time (minutes)	
	Mean	σ
FF	4246.5	14913.6
LML	4053.5	13570.0
LCL	4240.1	14906.1
LCD	289.5	1078.8
MCL	4247.2	14913.2
VCPU	4844.5	16725.9
PS	3895.0	12910.1

Table 9: Average server CPU steal time and CPU lost time of the different schedulers for the Bitbrains trace.

4.8.1 Performance of the Reflection Component

MF11 The reflection component produces policies that improve over past selection moments.

MF12 Improvements on past selection moments are not necessarily representative for potential improvements in the future.

The Bitbrains workload trace resulted in 87 selection moments for the portfolio scheduler and thus in 184 system state snapshots. Through genetic exploration the reflection component tries to find a scheduler configuration that improves the energy efficiency the most over the first 43 of these snapshots for the simulated duration. This cut-off is at 22,840 minutes in the trace, we display this point in the simulation in relevant figures as a dashed vertical red line. We want to see if based on historic reflection, the generated policy is selected and improves the future performance of the scheduler. This is why we include the generated policy from the reflection component after the moment of reflection.

Figure 25 depicts the genetic search process for the historic system state snapshots for the Bitbrains workload. We see that the final best scheduler configuration increases the power efficiency metric summed over all the simulated snapshots by 1.1 (MF11).

Figure 24 shows the overall power efficiency of the adapted portfolio, compared to the baseline portfolio. After the reflection point, we see no difference in overall power efficiency compared to the baseline portfolio.

Figure 26 depicts the selected policies for the extended portfolio. We see that the resulting scheduler configuration is being selected almost 35% of the time, yet there is no noticeable difference in power efficiency. An explanation may come from the incoming VM requests graph. Figure 27 depicts the pending VM requests. We see that 1,159 out of the 1,250 arrive at once at the start of the trace, leaving only a small portion of the scheduling decisions for after the moment of reflection. This could dominate the genetic search of the reflection stage, but when replaying the snapshot of this selection moment with the result of the genetic search, it does not improve this large selection moment, meaning it *was* the result of the smaller selection moments. The genetic search therefore was not dominated by this spike, yet still failed to improve future selection moments (MF12). After inspecting the scores of the individual policies during selection moments after the reflection stage, we see that for only 2 of the 30 times the reflected policy is selected, there is a small difference with the second best policy. The other 28 times they score the exact same, resulting in the same overall energy efficiency at the end of the trace. More frequent reflection could potentially adapt the portfolio towards a more diverse portfolio capable of improving future selection moments as well.

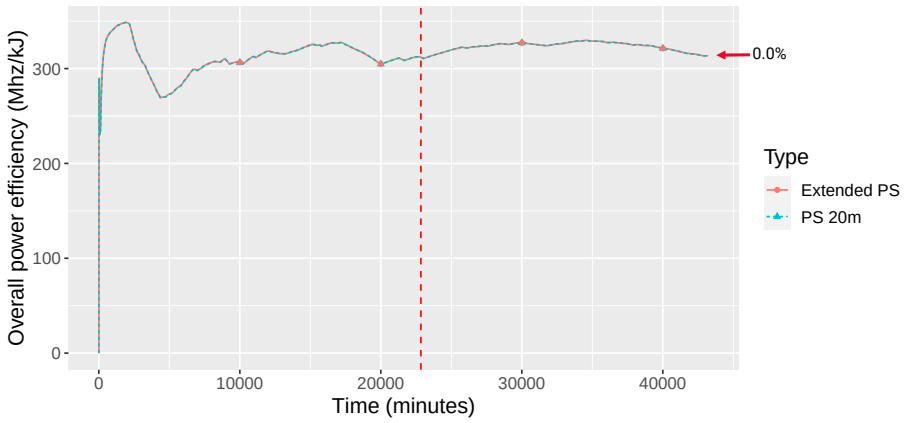

Figure 24: Overall power efficiency of the baseline portfolio compared to the adapted portfolio for the Bitbrains trace.

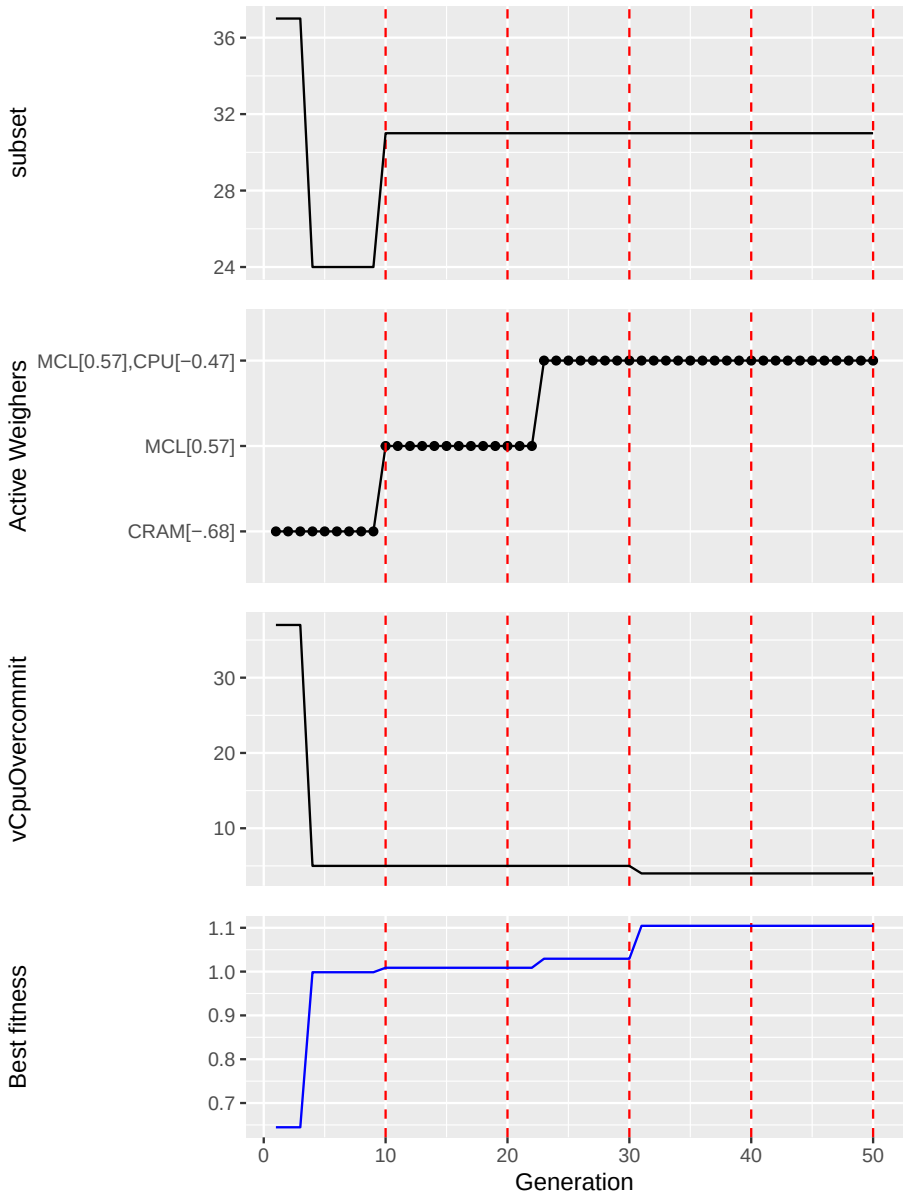

Figure 25: Elite fitness and scheduler configuration for the genetic exploration per generation for the Bitbrains snapshots. **MCL = MaximumConsolidation-Load Weigher, CRAM = CoreRam Weigher, CPU = CPU Load Weigher.*

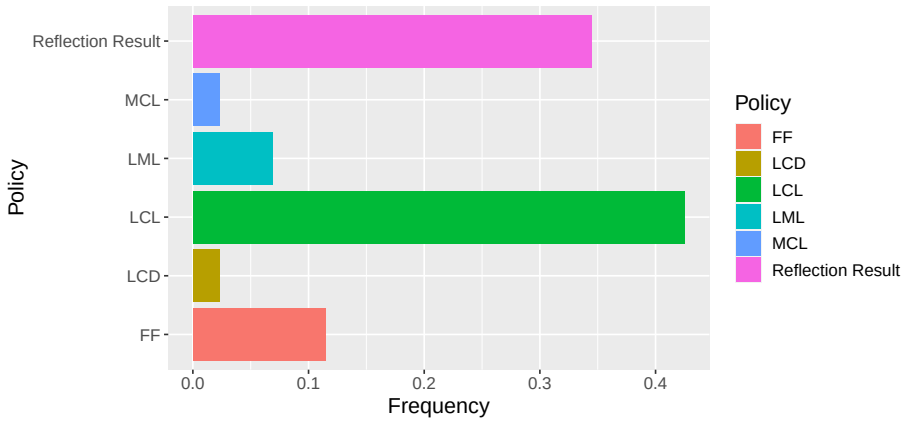

Figure 26: Policy selection frequency of the extended portfolio for the Bitbrains trace.

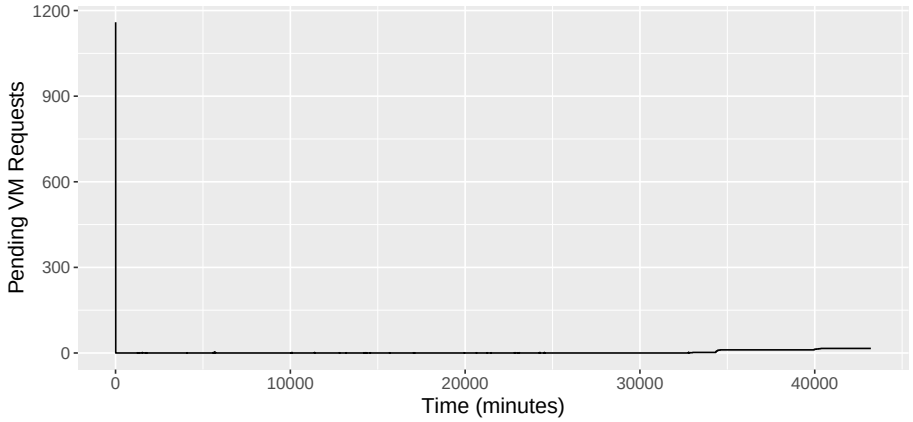

Figure 27: Pending VM requests over time for the Bitbrains trace.

4.9 Discussion

We now summarize the contributions of this chapter and discuss possible threats to their validity.

4.9.1 Summary

We present a working prototype of a portfolio scheduler with a reflection component capable of finding policies that are selected in future scheduling decisions,

based on historic performance data and a genetic search algorithm. We also extend previous portfolio schedulers with the ability to select a policy based on an energy efficiency based utility function. We show the impact of simulation duration, workload, and adaptations based on the implemented reflection stage on the performance of the portfolio scheduler.

4.9.2 Threats to validity

Internal Validity
The configuration of the evolutionary algorithm parameters is a threat to the internal validity of the results, as finer parameter tuning of the algorithm could lead to different results.

Construct Validity
The validity of the results produced by the portfolio scheduler could pose a threat to the validity of the results. We addressed this threat by writing a set of unit tests used to validate the portfolio scheduler simulator, which are released together with the source code for the experiments. The moment of reflection for the reflection component could also be a threat to the validity of the results, it is possible that different moments of reflection produce different results.

External Validity
The degree to which the workloads used are representative of the workloads of other datacenters running business-critical workloads is a threat to external validity. We addressed this threat by using multiple real-life business-critical workload traces gathered. Another threat to external validity is that we did not conduct experiments with scheduling policies that are energy-aware, as the portfolio of policies are based on non-energy aware policies.

5 Conclusion and Directions for Future Work

In this chapter we recap the contributions made in this thesis and discuss what directions of future work the results could lead to.

5.1 Conclusion

We investigate in this thesis two main research questions regarding a more generally capable energy efficient portfolio scheduler. We describe the problems with portfolio scheduling and energy efficient scheduling in Chapter 1 and provide background on the topic in Chapter 2. We present in Chapters 3 and 4 the main contributions of this thesis, by designing, implementing and evaluating a prototype to address the problem. We now answer the two main research questions:

(RQ1) Design: How can we design a more generally capable energy aware portfolio scheduler?

We have formulated requirements of a design for a more generally capable energy aware portfolio scheduler, after identifying a set possible stakeholders and use cases. From these requirements we have created a design of a portfolio scheduler with a novel reflection stage capable of suggesting adaptions to the portfolio that are likely to be selected in the future, based on historic performance data and system states. The designed portfolio scheduler also takes into account a utility function based on server-level energy efficiency in the selection of a policy.

(RQ2) Evaluation: How can we evaluate a self-reflective energy aware portfolio scheduler?

We implemented a working software prototype of the designed portfolio scheduler in OpenDC. We extended OpenDC with the capability to save and load system state snapshots, and in turn with the ability to use a portfolio scheduler with an arbitrary set of scheduling policies. We used real-life business-critical workload traces to conduct experiments on the performance of the portfolio scheduler regarding its ability to choose energy efficient policies and on the ability of the reflection component to suggest portfolio additions that are likely to be chosen in the future based on historic data. We find that while the genetic search is able to find improvements on past selection moments of the portfolio scheduler, this is not a clear indication of the level of improvements for future selection moments. Furthermore, we find that the simulated duration is not always representative for future performance for the given long-running workloads and can even lead to worse scheduling decisions in the long-term. We also find that selecting a policy on the work done per unit of energy alone, impacts other common performance metrics, signalling the need for a utility function that balances both energy-efficiency and common performance metrics.

5.2 Directions for Future Work

Based on the results and contributions presented in this work, we envision future areas of future work:

1. We see possibilities for *investigating VM migration* in combination with a portfolio scheduler for business-critical workloads. This could help alleviate poor long-term scheduling decisions taken by a portfolio scheduler, by reflecting upon individual VM performance and migrating them to other physical machines if the performance passed a certain threshold. This way the short-term performance gains of the portfolio scheduler are kept while bad placements are balanced in the long run.

2. To find a better balance between energy efficiency and more common performance metrics in schedulers, *investigation of a more balanced utility function* for a portfolio scheduler could lead to energy efficient schedulers that sacrife less in terms of these common performance metrics. This could improve adoption of energy efficient schedulers, as high performance is still important to datacenter customers.

3. *Experimentation with other types of workloads* from business-critical workloads, with more CPU and memory performance metrics could offer more meaningful results for the implemented portfolio scheduler.

4. *Further analysis of the implemented frequent reflection component* regarding other utility functions and more frequent periods of reflection could help improve the reflection abilities.

5. *Research into design space exploration of schedulers* could help improve the ability for the evolutionary algorithm to find schedulers that improve the portfolio. This includes different and more complex encoding of the schedulers.

6. *Experiments with energy-aware scheduling policies* added to the portfolio scheduler would help compare the performance of the portfolio scheduler to these energy-aware policies and could potentially improve the portfolio scheduler.

References

[1] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. **Recalibrating global data center energy-use estimates**. *Science*, **367**(6481):984–986, 2020.

[2] Cisco Global Cloud Index. **Forecast and methodology, 2016–2021 white paper**. *Updated: February*, **1**, 2018.

[3] Anders SG Andrae and Tomas Edler. **On global electricity usage of communication technology: trends to 2030**. *Challenges*, **6**(1):117–157, 2015.

[4] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. **An experimental performance evaluation of autoscaling policies for complex workflows**. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 75–86, 2017.

[5] David Villegas, Athanasios Antoniou, Seyed Masoud Sadjadi, and Alexandru Iosup. **An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds**. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 612–619. IEEE, 2012.

[6] Vincent van Beek, Giorgos Oikonomou, and Alexandru Iosup. **Portfolio Scheduling for Managing Operational and Disaster-Recovery Risks in Virtualized Datacenters Hosting Business-Critical Workloads**. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 94–102, 2019.

[7] Shenjun Ma, Alexey Ilyushkin, Alexander Stegehuis, and Alexandru Iosup. **Ananke: A Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows**. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 227–232, 2017.

[8] Kefeng Deng, Junqiang Song, Kaijun Ren, and Alexandru Iosup. **Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds**. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[9] Ohad Shai, Edi Shmueli, and Dror G Feitelson. **Heuristics for resource matching in intel's compute farm**. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 116–135. Springer, 2013.

[10] Harry M Markowitz. *Portfolio selection*. Yale university press, 1968.

[11] Prateek Sharma, David Irwin, and Prashant Shenoy. **Portfolio-driven resource management for transient cloud servers**. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, **1**(1):1–23, 2017.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. **Xen and the art of virtualization**. *ACM SIGOPS operating systems review*, **37**(5):164–177, 2003.

[13] James M Kaplan, William Forrest, and Noah Kindler. **Revolutionizing data center energy efficiency**. *McKinsey & Company*, pages 1–13, 2008.

[14] **Virtualization vs Cloud: Which is better for you?** `https://techgenix.com/virtualization-vs-cloud-which-is-better-for-you/`. Accessed: 30-08-2022.

[15] Sheng Di, Derrick Kondo, and Walfredo Cirne. **Characterization and comparison of cloud versus grid workloads**. In *2012 IEEE International Conference on Cluster Computing*, pages 230–238. IEEE, 2012.

[16] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. **The datacenter as a computer: An introduction to the design of warehouse-scale machines**. *Synthesis lectures on computer architecture*, **8**(3):1–154, 2013.

[17] Lotfi Belkhir and Ahmed Elmeligi. **Assessing ICT global emissions footprint: Trends to 2040 & recommendations**. *Journal of cleaner production*, **177**:448–463, 2018.

[18] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. **Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing**. *Future generation computer systems*, **28**(5):755–768, 2012.

[19] Juan Carlos Salinas-Hilburg, Marina Zapater, José M Moya, and José L Ayala. **Energy-aware task scheduling in data centers using an application signature**. *Computers & Electrical Engineering*, **97**:107630, 2022.

[20] Vincent Van Beek, Giorgos Oikonomou, and Alexandru Iosup. **Portfolio scheduling for managing operational and disaster-recovery risks in virtualized datacenters hosting business-critical workloads**. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 94–102. IEEE, 2019.

[21] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent Van Beek, and Alexandru Iosup. **OpenDC 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters**. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 455–464. IEEE, 2021.

[22] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The atLarge vision on the design of distributed systems and ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019.

[23] Fabian Mastenbroek. **Radice: Data-driven Risk Analysis of Sustainable Cloud Infrastructure using Simulation**. 2022.

[24] V Dinesh Reddy, Brian Setz, G Subrahmanya VRK Rao, GR Gangadharan, and Marco Aiello. **Metrics for sustainable data centers**. *IEEE Transactions on Sustainable Computing*, **2**(3):290–303, 2017.

[25] **SPECpower_ssj2008 benchmark**. `https://www.spec.org/power_ssj2008/results/res2022q1/power_ssj2008-20211221-01146.html`. Accessed: 24-08-2022.

[26] Michael Cardosa, Madhukar R Korupolu, and Aameek Singh. **Shares and utilities based power consolidation in virtualized server environments**. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 327–334. IEEE, 2009.

[27] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-scale cluster management at Google with Borg**. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[28] Vincent van Beek, Jesse Donkervliet, Tim Hegeman, Stefan Hugtenburg, and Alexandru Iosup. **Self-expressive management of business-critical workloads in virtualized datacenters**. *Computer*, **48**(7):46–54, 2015.

[29] Arman Shehabi, Sarah J Smith, Eric Masanet, and Jonathan Koomey. **Data center growth in the United States: decoupling the demand for services from electricity use**. *Environmental Research Letters*, **13**(12):124030, 2018.

[30] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. **Worth their watts?-an empirical study of datacenter servers**. In *HPCA-16 2010 The Sixteenth Inter-*

national Symposium on High-Performance Computer Architecture, pages 1–10. IEEE, 2010.

[31] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[32] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. **A reference architecture for datacenter scheduling: design, validation, and experiments**. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 478–492. IEEE, 2018.

[33] **Jenetics: Java Genetic Algorithm Library**. `https://jenetics.io/`. Accessed: 12-08-2022.

[34] Brad L Miller, David E Goldberg, et al. **Genetic algorithms, tournament selection, and the effects of noise**. *Complex systems*, **9**(3):193–212, 1995.

[35] Tobias Blickle and Lothar Thiele. **A comparison of selection schemes used in evolutionary algorithms**. *Evolutionary Computation*, **4**(4):361–394, 1996.

[36] Vincent Van Beek, Giorgos Oikonomou, and Alexandru Iosup. **A CPU contention predictor for business-critical workloads in cloud datacenters**. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 56–61. IEEE, 2019.

[37] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick HJ Epema. **The grid workloads archive**. *Future Generation Computer Systems*, **24**(7):672–686, 2008.

[38] Siqi Shen, Vincent Van Beek, and Alexandru Iosup. **Statistical characterization of business-critical workloads hosted in cloud datacenters**. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474. IEEE, 2015.

[39] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. **An analysis of performance interference effects in virtual environments**. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 200–209. IEEE, 2007.