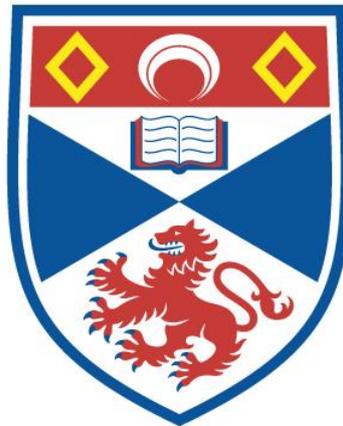


INVESTIGATING PERFORMANCE AND ENERGY EFFICIENCY ON A PRIVATE CLOUD

James William Smith

**A Thesis Submitted for the Degree of PhD
at the
University of St Andrews**



2013

**Full metadata for this item is available in
Research@StAndrews:FullText
at:**

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/6540>

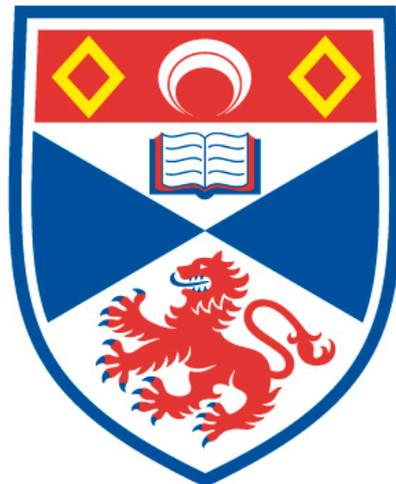
This item is protected by original copyright

**This item is licensed under a
Creative Commons Licence**

Investigating Performance and Energy Efficiency on a Private Cloud

James William Smith

PhD Thesis



University of
St Andrews

A thesis submitted to the University of St Andrews

for the degree of Doctor of Philosophy

September 2013

Declaration

I, James William Smith, hereby certify that this thesis, which is approximately 50,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in September 2009 and as a candidate for the degree of Ph.D. in September 2009; the higher study for which this is a record was carried out in the University of St Andrews between 2009 and 2013.

Date Signature of candidate

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Ph.D. in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date Signature of supervisor

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis:

(i) Access to printed copy and electronic publication of thesis through the University of St Andrews.

Date Signature of candidate

Date Signature of supervisor

Abstract

Organizations are turning to private clouds due to concerns about security, privacy and administrative control. They are attracted by the flexibility and other advantages of cloud computing but are wary of breaking decades-old institutional practices and procedures. Private Clouds can help to alleviate these concerns by retaining security policies, in-organization ownership and providing increased accountability when compared with public services.

This work investigates how it may be possible to develop an energy-aware private cloud system able to adapt workload allocation strategies so that overall energy consumption is reduced without loss of performance or dependability.

Current literature focuses on consolidation as a method for improving the energy-efficiency of cloud systems, but if consolidation is undesirable due to the performance penalties, dependability or latency then another approach is required.

Given a private cloud in which the machines are constant, with no machines being powered down in response to changing workloads, and a set of virtual machines to run, each with different characteristics and profiles, it is possible to mix the virtual machine placement to reduce energy consumption or improve performance of the VMs.

Through a series of experiments this work demonstrates that workload mixes can have an effect on energy consumption and the performance of applications running inside virtual machines. These experiments took the form of measuring the performance and energy usage of applications running inside virtual machines. The arrangement of these virtual machines on their hosts was varied to determine the effect of different workload mixes.

The insights from these experiments have been used to create a proof-of-concept custom VM *Allocator* system for the OpenStack private cloud computing platform. Using *CloudMonitor*, a lightweight monitoring application to gather data on system performance and energy consumption, the implementation uses a holistic view of the private cloud state to inform workload placement decisions.

Personal Acknowledgments

I must begin by thanking my supervisor Prof. Ian Sommerville for his invaluable guidance and endless patience. Best wishes on your retirement.

Dr. Lee Cadman and the Software Team at McLaren, thank you for pushing me to be a better engineer.

Angus, Greg, Ali, David, Lukasz, Derek, Jonathan, Gordon, Graham & the others in the School of Computer Science and LSCITS who I've had the pleasure to work with and learn from. Thank you for your kindness and support.

My family, and in particular my parents and grandparents, you are always my motivation and I am deeply grateful for the opportunities you gave me.

To my friends, especially Darren, Jack, Paul, Phil, Rob, Tom and V, thanks for giving me welcome distractions and constant encouragement.

Finally, thanks to the University and town of St Andrews. I feel so fortunate to have been a part of this community.

Table of contents

1	Introduction	1
1.1	Project aims and motivations	5
1.2	Novel contributions	5
1.3	Thesis structure	8
2	Literature review	10
2.1	Introduction	10
2.2	Energy measurement.....	18
2.3	Workload mixing	30
2.4	Modifying OpenStack	37
2.5	Research methods	42
3	Software based energy measurement	44
3.1	Introduction	44
3.2	Power model-based estimation.....	46
3.3	<i>CloudMonitor</i> evaluation	62
3.4	Conclusion	75
4	Workload mixing	77
4.1	Introduction	77
4.2	Experimentation.....	79
4.3	Discussion & conclusion	104
5	Implementation of OpenStack <i>Allocator</i>	106
5.1	The OpenStack project and system architecture	108
5.2	<i>Allocator</i> architecture	112

5.3	Rules language	114
5.4	<i>Filter</i>	117
5.5	<i>Weigher</i>	125
5.6	Data store and decision tracking	133
5.7	Tracking termination of VMs	136
5.8	Deployment and use.....	137
6	Evaluation of OpenStack <i>Allocator</i>	139
6.1	Introduction	139
6.2	Experimental design.....	140
6.3	Alternative workload allocation strategies	149
6.4	Experimental results.....	153
6.5	Conclusions.....	171
7	Conclusions and future work.....	177
7.1	Critical evaluation of the work done.....	179
7.2	Discussion	184
7.3	Future work	185

1 Introduction

Cloud Computing, a model for delivering self-service, on demand computing resources, is changing the face of the IT industry. The US National Institute of Standards and Technology (NIST) definition of cloud computing [1], one of the most widely accepted, describes the area as having:

- **Essential characteristics:** on-demand self-service, broad network access, resource pooling, rapid elasticity and measured services.
- **Deployment models:** private, community, and public clouds.
- **Service models:** Software, Platform and Infrastructure as a Service.

Almost every major corporation in the IT industry now has some interest the area of cloud computing. These companies are leading the way in technological advances; in recent years we have seen the launch of a number of public cloud services such as Amazon EC2, Google App Engine and Microsoft Azure. They are investing large amounts of capital and personnel to research ways of leveraging the emerging market that analyst firm *The 451 Group* suggests will grown to \$20bn by 2016 ¹.

Cloud computing has been described by a group of industry Chief Technological Officers as “*a fundamental change in how IT is provisioned and used*” [2]. Now, any organization can dynamically scale their resources according to how much they require and billing can be done on a finer grain than traditional platforms where the procurement of an entire physical server would be the single smallest billable item. With a cloud platform the use of individual resources can be billed; from CPU cores used to GBs of data transferred in I/O operations.

There are three types of Clouds. A ‘public cloud’ is one where services are provided to external users. They require no capital investment from the users,

¹<http://news.techworld.com/virtualisation/3464992/public-cloud-revenues-to-hit-20-billion-by-2016-says-451-research/?olo=rss>

moving this cost and risk to the providers. The service usually provides fine-grained control over services and billing allowing users to pay for their computing as a utility. They are however vulnerable to issues relating to privacy, security and administrative control. 'Private clouds' provide control over some of the issues that can stop organisations using public providers, but do not provided the same lack of capital expenditure or flexibility. 'Hybrid clouds' is the term given when an organisation uses a combination of the above models, for example initially using a private cloud but bursting out to a public cloud service should demand exceed the in-house infrastructure.

In recent years, there has been an uptake in the number of organizations pursuing private cloud computing installations within their organizations. There are number of reasons for this, but primarily organizations want the flexibility and dynamic provisioning benefits of cloud within the regulatory, security and administrative polices and procedures that they already have in place.

Cloud computing is generally provided through datacentres with hundreds or thousands of physical servers. It is estimated that datacentres consume up to 1.5% of all the generated electricity in the world [3], a figure that is likely to grow as cloud usage becomes widespread, particularly as these systems are "always-on always-available". However, the datacentres required by clouds also have the potential to provide the most efficient environments for computation, as tasks and computers to run them can be consolidated in purposefully designed buildings that have been architected to minimise expenditure on non-computational energy consumption such as cooling and lighting.

When data centres expand or new data centres are built the number of machines consuming electricity will increase. However cloud computing has particular advantages over traditional IT systems that can be exploited to reduce the amount of energy consumed.

The key technological enabler of cloud computing is virtualization, a technology which allows physical servers to run at a higher level of utilization by running more than one virtual machine on each physical server. A datacentre that previously had 10,000 physical machines running at an

average utilization of 20% may be able to reduce the number of physical machines to 2,500 running 10,000 virtual machines for an average physical server utilization level of 80%. In practice the machines may not be as heavily consolidated as this, but the potential for energy savings are clear. This will be illustrated in the following calculation.

Each electrical appliance, including computing servers, consumes electricity to do work. The average amount of power drawn for an appliance during its use is known as the “average wattage rating”. For a high-end computing server, this rating would be about 200W².

Running a 200W appliance for 24 hours a day, 365 days a year would result in 1752 kWh being consumed using this formula to convert watts to kilowatt-hours:

$$E_{(\text{kWh})} = P_{(\text{W})} \times t_{(\text{hr})} / 1000$$

Currently, the University of St Andrews is charged £0.089 per kWh for electricity³. The yearly charge for 1752 kWh would therefore be £155.93.

If an organisation has 10,000 physical servers at an average wattage rating of 200W they would spend approximately £1.56 million per year on electricity. Over 5 years, this will amount to £10.5 million including a 15% per annum rising fuel cost, which has historically been the amount by which the electricity at costs at St Andrews have risen.

Despite organisations facing rising costs driven by energy, research from the UK's National Computing Centre [4] suggest that only 13.4% of UK organizations currently monitor their energy consumption. Providing administrators with data about energy usage is the first step to promoting an energy aware culture and tackling reduction in consumption.

Improving understanding about the energy usage of private cloud

² This number is taken from the Dell PowerEdge R610 servers used in this work. It is an illustrative example.

³ As of July 2013.

installations is the first step in making these systems energy-aware. The insights gained from instrumenting these systems can then be used to develop new strategies that will reduce their energy consumption.

In order for organisations to widely adopt energy saving techniques, energy management techniques should not compromise application performance or system dependability. Current literature in the field of energy efficient cloud computing tends to focus on the reduction of energy at all costs, neglecting other issues that are important to organisations.

This thesis describes work on enabling instrumentation of energy usage and providing a mechanism to reduce the power usage of servers in private cloud systems. Our approach has been to develop software to reduce the energy consumption of organizations running a private cloud and then evaluate the effectiveness of this software through experimentation. The software applications that have been developed are:

- An energy monitor: this application, *CloudMonitor*, provides a means to monitor resource consumption and energy usage at scale. This is achieved using the module to directly interact with power measurement hardware or through estimation of the energy load using the resource monitoring metrics and a mathematical power model.
- An energy-efficient workload allocation system: The *Allocator* software was developed after experimenting with and analysis of different workload mixes on a typical private cloud system. The lessons from this analysis were incorporated into a workload allocation strategy for OpenStack, a popular open-source private cloud computing platform.
- A tool for running energy measurement experiments on a private cloud: the *ExperimentRunner* tool has been developed to thoroughly test and evaluate the effectiveness of the *Allocator* software through a series of experiments.

Using these tools together, with the monitoring tool used to provide the information necessary for the *Allocator* to be effective in its distribution of workload, enables organizations to effectively monitor their energy usage and

take steps to reduce their consumption.

1.1 Project aims and motivations

The aim of this project is to provide a means to measure and potentially reduce the energy consumption used by private cloud computing systems within organizations. Our work, therefore, has two broad objectives:

1. Develop a scalable mechanism for measuring the energy consumption of shared servers in a cloud environment
2. Build a virtual machine allocation strategy for a private cloud system that reduces energy usage without compromising application performance or dependability.

There are two main motivations that have driven this project. Firstly, the rise of green computing is a topical issue so there is significant interest from organizations to reduce their energy costs and eliminate carbon emissions. Secondly, the adoption of cloud computing in recent years leads to a new set of challenges for green computing researchers, as the underlying technology changes so must the techniques by which systems efficiency is improved.

The challenges that organizations face are firstly, to understand the scale and profile of their energy usage, particularly where resources are currently being used and what the immediate steps are that can be taken to reduce those costs.

Secondly, if it is possible to gain insight from private cloud platforms resource usage, there are improvements that can be made to how that system is operated. It may be possible to improve the operational procedures of the system to reduce energy consumption.

The research that is presented in this thesis attempts to address these challenges.

1.2 Novel contributions

There are three novel contributions made by this work:

1. Scalable, software-based energy monitoring. The development of the *CloudMonitor* tool, that allows the measurement of energy consumption through software. *CloudMonitor* is based on the development of a power model to determine energy usage without requiring a dedicated power meter for each compute node.

A power model is a mathematical model that describes the relationship between subcomponent use and the amount of energy consumed. These are useful tools for estimating energy consumed by a server or virtual machine without the expense of dedicated power measurement hardware. They have been demonstrated to be accurate with an error rate of up to 6% [5][6][7][8][9].

CloudMonitor improves on the current state of the art by basing its power models on an accurate billing-level Power Distribution Unit (not, like some of the literature, using an consumer off-the-shelf device). The power model is derived automatically and has been shown to be accurate to up to 96%.

Power usage measurement is easily scalable using *CloudMonitor* as the software can develop a model to fit a particular server configuration and then be rolled out across all servers of the same configuration in a datacentre to report on their energy usage with additional dedicated power measuring devices.

The software is written in Java so is portable across most common operating systems including Linux, Windows and Mac OS X.

2. An understanding of the relationships between energy consumption and application performance when arranging tasks in a virtualized environment. The workload mix experiments in Chapter 4 present a look at how reconfiguration of server workload can lead different energy consumption levels and impact application performance.

The current literature notes, “...*there has been little work done on joint power and performance aware schemes for multi-dimensional resource*

allocation” [10]. Most current energy-efficient schemes for allocating workload are concentrated on reducing energy usage at all costs, sometimes to detriment of application performance [11]. Those that do consider application throughput are either concerned with the performance of only one resource [12] or in the reduction of VM interference [13].

The experiments of Chapter 4 show that energy savings can be achieved while limiting the impact to application performance.

3. Demonstration of a strategy for allocating virtual machines to cloud servers that reduces energy consumption without significant performance penalties. The insights from the workload mix experiments were used to develop a new VM *Allocator* for OpenStack based around co-location of compatible VM types. Users were requested to provide information on the type of resources their VM would use, and this meta-data was used in the allocation decision.

Current work on developing an energy aware VM allocation system for OpenStack has focused on consolidation of VMs on to the fewest number of physical hosts as possible [14][15][16][17] without concern for application performance.

Our work shows that the developed *Allocator* can reduce the energy spent on a standard OpenStack installation by around 10%, compared to between 6-13% for an alternative strategy that is only concerned with consolidation to save energy. The *Allocator* can also achieve higher levels of application performance, for example, sometimes it can double the I/O throughput of the alternative energy efficient strategy.

The *Allocator* software is open source and can easily be used to modify a standard OpenStack installation.

The work in this thesis has been included in 3 publications, the details of which are included in Appendix D.

1.3 Thesis structure

This thesis is made up of seven chapters, which are outlined below:

Chapter 2 (*Literature Review*) discusses the current state of research in each of the areas relevant to this thesis, including a discussion of green cloud computing, power monitoring and allocation strategies to reduce energy consumption. It concludes with an analysis of the limitations of existing systems and provides motivation for the work undertaken.

It is clear from the current state of the literature that there is scope for the development of a virtual machine allocation system on OpenStack that limits the amount of energy consumed while attempting to preserve the performance of applications.

Chapter 3 (*Software based Energy Measurement*) describes work on estimating the energy usage of distributed computation systems, including the design, implementation and evaluation of the tool *CloudMonitor*, an accurate, scalable, and portable system for measuring energy on a cloud computing infrastructure system. *CloudMonitor* can estimate energy usage using a mathematical power model that is calibrated against a hardware power measurement device. Once trained, the tool can estimate energy usage for any number of servers of the same configuration.

Chapter 4 (*Workload Mixing*) Workload mixing is introduced as a method for organising workload on a set of physical hosts so that desired characteristics can be exploited. This chapter identifies the trade-offs between energy usage and application performance in a virtualized in environment through a series of experiments. These experiments show that workload mixes affect application performance and system energy usage for the hardware and workload used. The results lead to the possibility of an optimal workload mix for each situation, something that could be adapted into a *VM Allocator* for a private cloud system.

Chapter 5 (*Implementation of OpenStack Allocator*) discusses the design, implementation and evaluation of an energy-aware *VM Allocator* for the OpenStack private cloud computing system. Drawing on conclusions from the

previous chapter, the *Allocator* is implemented to reduce the energy an organization requires to operate their private cloud installation without compromising performance. The *Allocator* software requires users to label the VMs they are requesting in terms of the hardware resource they will use the most, information that along with current usage metrics in the system provided by *CloudMonitor*, allows the algorithm to make effective placement decisions.

Chapter 6 (*Evaluation of OpenStack Allocator*) evaluates the *Allocator* from Chapter 5 with a series of experimentations comparing the energy usage and application performance of the *Allocator* and other alternative VM placement strategies. It was found that the *Allocator* could achieve around 10% reduction in energy usage in most situations, while the alternative energy-efficient strategy fluctuates between 6-13% reduction with more significant impacts on application performance.

Chapter 7 (*Conclusions & Future Work*) concludes the thesis by discussing the points raised by this work and providing an evaluation of the software tools developed. Future work that could expand on the work of this thesis is also described, including the benefits that could be potentially exploited from the added layer of interaction between users describing their workloads and the private cloud system.

2 Literature review

2.1 Introduction

In 2007, the total carbon footprint of the IT industry, including personal computers, mobile phones, telecom devices and data centres was estimated to be 830 Metric megatons of carbon emissions (MtCO₂), 2% of the estimated total emissions from all human activity that year. This figure is expected to grow as the use of IT equipment continues to rise [18].

Yamini *et al* [19] describe the field that attempts to reduce the energy consumption of IT equipment:

“Green computing is the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated sub systems such as monitors, printers, storage devices, and networking and communications systems efficiently and effectively with minimal or no impact on the environment.”

The term “Green Computing” came from the US Environmental Protection Agency that launched their Energy Star program in 1992 to label energy-efficient equipment that is sold in the United States. The Swedish Organization TCO, who developed the TCO certification program with the aim of reducing the energy consumption of IT equipment and eliminating hazardous materials used in their construction, started a similar movement [20].

For most organizations, carbon emissions are a serious concern, and one that will grow in the coming years as worldwide governments apply carbon taxes, but that concern is still secondary to the real world cost of owning and operating computing equipment.

Total cost of ownership (TOC) of a computing system is no longer simply dominated by the initial capital expenditure as operating costs are instead becoming a significant factor. Reports have estimated that the capital cost of

acquiring computing hardware is likely to be exceeded by the total cost of operation, mostly from the cost of electricity needed to operate and cool, even over a relatively short amortization period of 3-5 years [21][22][23]. A back of the envelope calculation shows that an organisation which is charged £0.089 per KWh⁴ will spend £155.93 per year on a computing device rated at 200W⁵ running 24 hours a day. Over 5 years, this will amount to £1051 with a fuel cost rising at 15% per annum⁶, close to the purchase cost of a new computing server. If organizations wish to reduce their expenditure on electricity, they will first need to understand their energy usage and gather data to begin tackling a reduction in consumption.

One of the first challenges organizations face in reducing energy consumption will be to gather sufficient data about their energy consumption through instrumentation. A UK survey found that only 13.4% of UK organizations measure their energy consumption [4]. Overcoming this challenge and providing decision makers with enough information to tackle reducing consumption should be the priority for the remaining organizations that do not instrument energy.

Green Computing spans the whole breadth of the computing world, from small mobile devices such as phones or tablets to the servers used in the data centres that power the world's cloud services. There are a number of different approaches to Green IT, at different levels of a computing system.

Firstly, at the individual machine level, efforts from vendors have resulted in chipset improvements like the latest generation of Intel *Haswell* processors that claim a power reduction of up to 50 percent compared to the previous generation of processors. Such reductions will result in longer battery life for mobile devices or lower power demands for desktop PCs and servers [24][25]. New technologies are also replacing older components in an individual machine, such the move towards solid state hard drives that provide increased

⁴ The amount in dollars that the University of St Andrews is charged for at the time of writing

⁵ A standard power rating of a high-end computing server

⁶ Average yearly rise in energy cost at the University of St Andrews.

hard drive performance and draw less energy, requiring only 13% of the power required for a comparable HDD in some cases [26].

On mobile devices approaches to reducing the energy consumption of standard tasks have resulted in different devices consuming different amounts of energy to do the same work. Mayo *et al* [27] discovered that even simple tasks such as listening to music, making a call, etc. could consume widely different amounts of energy on different devices. As they are basically the same task, this shows that manufactures have different approaches to optimizing their devices for energy efficiency.

At the data centre level, recent announcements from large Internet companies such as Google and Facebook suggests that breakthroughs have been found in mechanical engineering as their PUE, or energy usage Effectiveness, is now close to 1.0.

PUE is a metric for describing the efficiency of a data centre in terms of its electricity use. It aims to compare how much power is being used for useful computing and how much is needed for infrastructure. A data centre's PUE is the ratio of total energy consumed by the facility the to power used by the computing equipment. It is defined as:

$$\text{PUE} = \text{Total Facility Power} / \text{IT Equipment Power}$$

Total Facility Power is defined as the power used for the data centre (power not used for any other purposes, for example where the data centre is in a mixed-use building is not included) and IT Equipment Power is defined as the power required by all ICT equipment within the data centre.

In an ideal world 100% efficiency would be achieved to give PUE rating of 1.0 meaning all power is used by IT equipment only. Research from the Lawrence Berkley National Labs [28] shows that 22 data centres measured in 2008 have PUE Values in the range 1.3 to 3.0. Modern data centres such as those built by Google and Facebook claim PUE values in the range of ~1.1^{7,8}. Such efficiency

⁷ <http://www.google.com/about/datacentres/efficiency/internal/>

⁸ https://www.facebook.com/prinevilleDataCentre/app_399244020173259

can even be in the reach of smaller organizations like the University of St Andrews, which built a new data centre with a PUE of 1.2⁹.

PUE is effective at measuring the amount of power required above that used by the individual servers doing useful work. It details the power required for cooling, lighting and other infrastructure costs in relation to that used for the IT equipment. If the computing servers use 1kW of power, and 100 W is used by the cooling and lighting systems, then the data centres PUE would be 1.1¹⁰. In such situations there could be productive gains in focussing on reducing the amount of energy used by the computing systems, even if such a reduction would result in a higher PUE score because less energy is used for computing in relation to infrastructure.

Many organisations are trying to reduce their energy demands at the organisational level through policy changes. Administrators are requesting per building data about energy consumption, for example the University of St Andrews Energy Report website¹¹ that provides per building breakdown of energy costs and usage analysis. With increased awareness of energy demands, administrators can begin to enact energy aware IT policies, from mandating the shutting down of machines when not in use to the implementation of energy efficient scheduling algorithms for data centres.

In recent years, there has been increased focus at the organisational level on carbon emissions and that steps that can be take to make an organisation carbon neutral. For example, the Guardian newspaper, in conjunction with academic researchers, has recently taken steps to estimate the carbon impact of the consumption of their online services [29]. The results of this estimation concluded that:

“The carbon footprint of the online newspaper amounts to approximately 7700 tCO₂e per year, of which 75% are caused by the user devices”.

⁹

http://www.goodcampus.org/uploads/DOCS/111Case_4_st_andrews_Data_Centre_v4.pdf

¹⁰ Illustrative example

¹¹ <http://energy.st-andrews.ac.uk>

The work also evaluates different intervention scenarios based on steps that The Guardian could take to reduce their overall footprint through policy, for example mandating lower bit rate videos or purchasing additional CDN (content delivery network) capability.

Other companies, some of which are not originally in the business of IT infrastructure, such as McLaren, are lending their expertise to improving the energy efficiency of data centres. The McLaren F1 team became the world's first carbon neutral Formula One team in 2011 [30] and since has partnered with technological company IO to bring its expertise in reducing carbon emissions to data centres [31].

Finally, individual users have also be targeted by Green IT researchers in attempts to influence behaviour by providing feedback as to how their actions affect the energy consumption of IT equipment. Yu *et al* [32] describe an approach to providing energy awareness feedback to students in a computing laboratory. Liikkanen [33] notes that end users rarely have a full grasp of their total energy consumption or steps that they can take to reduce that consumption, making feedback tools especially important.

The focus of this thesis is on the steps that organisations can take at the data centre level to monitor and reduce the energy consumption of computing servers as they do work. Chen *et al* [34][35] note:

“The growth of the cost of electricity consisting of server power and cooling power outpaces expectations. In 2011, U.S. data centres spent about \$7.4 billion in electric power among which server power and cooling power contribute significantly to the total.”

The United States Environmental Protection Agency (EPA) highlighted some key issues with regards to computing datacentres in the United States in their 2007 report: [36]

- U.S. Datacentres consumed 1.5% of the total electricity generated in the whole country that year, the equivalent of the combined consumption of 5.8 million average U.S. households or a cost of roughly \$4.5 billion dollars.

- The amount of total energy consumption in this area has doubled in the period 2000-2006 and equivalent growth is expected in the coming years.
- The report suggests that 50% of the energy consumed by datacentres can be attributed to the useful work done by the computing servers. The other 50% is expended on infrastructure needs such as cooling, lighting, and security. This would suggest an average PUE of 2.0. This may have improved in the subsequent years since the report, as shown by the low PUE claims of Google and Facebook for new data centres.

The Global e-Sustainability initiative report [37] that datacentres are thought to be responsible for around 80-116 Metric Megatons of Carbon emissions (MtCO²) each year world wide, a figure not unlike that of countries such as the Netherlands and Argentina (146 and 142 MtCO² respectively).

We are primarily concerned with computing servers in this work, because advances in mechanical engineering have lowered PUE and improved general data centre infrastructure efficiency. The next biggest gains are likely to be found by reducing the power required to perform computing operations especially as cloud computing increases demand for IT services. Clouds have three key software factors that could help to reduce the energy consumption in datacentres [39][40]:

- Dynamic Provisioning
- Multi-tenancy
- Server Utilisation

Dynamic provisioning allows users to request only the resources they require at any one time, avoiding unused resource waste. If computing resource had to be allocated without dynamic provisioning, this leads to provisioning for peak demand, resulting in inefficiencies concerning over-provisioning of hardware resources and therefore energy consumption would be greater than that which is actually required for the amount of work.

Multi-tenancy is enabled by virtualization technology that allows multiple

users to have virtual machines on the same physical host and therefore allow that physical host to be shared amongst more than one user. If a host is shared amongst users then its energy usage can be amortized and higher levels of individual machine utilization achieved.

If multiple virtual machines are on a single host, that host can achieve a higher level of utilization than if it was assigned to a single user. If hosts are more fully utilized, there can be a requirement for fewer physical servers. Meisner *et al* [41] have found that an idle server consumes approximately 60% of the peak power. A server that is on, but underutilized therefore still consumes a significant level of power, so a lower number of higher utilized servers could reduce energy costs if the unused machines are powered down or are not installed at all.

Earlier in this chapter it was calculated that the average computing server with a rating of 200W would cost approximately £156 per year in electricity. If this server were left idle for a year, rather than highly utilized, it would still cost £93.60 per year simply to have it powered on, perhaps provisioned for peak demand.

Efficient use of virtualization is a key technology that enables green cloud computing. Virtualization technology enables dynamic provisioning, multi-tenancy and higher server utilization. It allows more computing to be done with less hardware and presents challenges as to how best exploit its characteristics to efficiently arrange virtual machines. Approaches to allocating virtual machines to manage energy consumption are discussed in Section 2.2.2.

Studies have shown that there exists a definite relationship between energy consumption and system performance. In *The Case for Energy-Proportional Computing* [22], Barroso and Holzle present evidence that hardware components in a computing system have dynamic power ranges, which can be up to 70% for modern CPUs. At the low end of the power range, the component is consuming some amount of power while doing no useful work. When the performance begins to increase, from 0% up to 100% utilization the dynamic energy consumption also increased. The authors found that the dynamic power range of RAM is around 50%, 25% for DISKs and 15% for

networking switches which means those components in particular consume a large percentage of their peak power when completely idle.

Pinherio *et al* [42] found that application performance throughput dropped when fewer computing devices, and therefore less power, were used so care must be taken when consolidating to reduce power. Felter *et al* [43] also found that individual components scaled their energy consumption according to performance. When a CPU chip was processing more instructions per cycle or when memory bandwidth was increased, the components consumed more power.

These results confirm the relationship between power and performance of applications, as the applications are dependent on computing components. If an acceptable solution is to be found to reduce the energy consumption of a computing system it must balance the reduction in power with preserving application performance. Devising a means of allocating virtual machine workload in a private cloud computing system to reduce energy while maintaining performance is one of the key aims of this thesis and is discussed in Chapters 4 and 5.

2.1.1 Literature review structure

This thesis focuses on the challenges that organizations face when attempting to understand and then reduce the energy consumption that they spend on computing.

Section 2.1 of this chapter introduced the current state of green computing including advances in server and data centre efficiency.

The first part of this thesis discusses the measurement of energy consumption through software and presents the development and evaluation of a tool, *CloudMonitor*, which is designed to aid organizations in instrumenting their energy usage. Section 2.2 of this Chapter discusses the software based energy estimation literature and includes a review of other tools that attempt to measure energy consumption through specialized hardware devices and software.

Section 2.2 is structured to first look at traditional monitoring systems, particularly those that are used in large scale environments, before moving on to discuss the methods behind resource gathering and power measurement before concluding with a discussion of the current state of the art in power modelling for estimating energy usage.

The second part of this thesis discusses the effect of workload placement strategies on energy consumption and application performance. Section 2.3 of this chapter discusses the relevant literature for workload placement strategies to reduce energy usage.

The final part of this thesis focuses on the development of a custom workload placing mechanism for OpenStack. Section 2.4 discusses other academic research attempts at modifying OpenStack and similar private cloud systems to improve energy consumption and application performance.

Finally, this chapter ends with a brief review of the research methods that were used in our work, which included data analysis, quantitative experimentation and workload generation.

2.2 Energy measurement

The software based energy measurement literature that is relevant to this thesis can be divided into a number of categories including the traditional techniques monitoring computing systems, particularly large systems with thousands of compute nodes (Section 2.2.1), tracking the energy usage of computing systems (Section 2.2.2), and finally software based power models for energy estimation (Section 2.2.3).

For the purposes of this literature review the scope of energy measurement is limited to only that used for computing hardware and does not include other infrastructure that may consume energy, such as cooling or lighting.

2.2.1 Monitoring computing systems

The real-time utilization of system resources is obtained by the use of

hardware resource monitoring. Tools that provide this functionality aim to give users an insight into the *black box* of computing hardware, showing how much of each subcomponent resource is being used in real time and providing historical figures for analysis. This can be very useful for system administrators to investigate how their systems are performing and help alert them of any issues that have arisen.

In hardware resource monitoring, metrics are normally provided about the physical machine subcomponents, such as: CPU, memory, hard disk and network usage. The collection of such information can be scaled to include data from a large number of machines. Such data collection has been used in grid computing and other shared systems, where users can be billed on the resources they consume. In cloud computing, the pay-as-you-go service model is enabled through this collection of resource usage data.

Zanikolas *et al* [44] present a discussion of monitoring tools and their different architectures in the context of grid computing. Some of the tools described, such as *Ganglia* [45] provide the ability to programmatically analyse trend information to gain greater insights into the system. *Ganglia* also presents the monitoring data in an easy to access graphical web interface to give users and administrators a quick overview of system behaviour.

The authors of *Ganglia* [45] describe key design challenges for distributed monitoring systems including scalability, overhead or load, and portability. To enable monitoring at scale, software architectures are required that allow data to be produced by sensors monitoring a large number of machines and allow that data to be collected, stored and analysed.

In *A Performance Study of Monitoring and Information Services for Distributed Systems* [46] the authors evaluate the performance of three monitoring systems: MDS2 [47], R-GMA [48] and *Hawkeye* [49], part of the *Condor* system. Each of the systems profiled was evaluated to have good scalability, perhaps due to their shared software architecture of three main system components: an information collector, information server and a directory where a list of the systems under observation is maintained.

Such a hierarchal architecture allows data from a sensor or information

collector to be communicated to other parts of the software system by a producer component that serves the data. Modules that receive and use this data, such as a trend analysis engine, are labelled as consumers. They can subscribe to data they are interested in using a publisher/subscriber model. A pub/sub model aids the scaling of a system by restricting information flow to only that which is required. These architectures make it possible for monitoring information to be collected and analysed for a large number of machines.

Chung *et al* [50] note that there is a “*trade-off between information fidelity and transmission cost*”. The more frequent and detailed the information being passed between components of a monitoring system the higher the cost of the transmission. Overhead of a monitoring system is a significant factor in its effectiveness as the *observer effect* will be felt if the load unduly impacts the performance of the components it is monitoring and as such compromises the reliability of the data. Lucas [51] notes, “*even though software measurement may affect the process being monitored, the degradation of useful compute time can be kept to a minimum*”. Keeping overhead at a minimum to preserve the integrity of the collected data and maintain system performance has to be one of the key aims when designing a monitoring system.

As monitoring systems may have to be deployed in heterogeneous environments, portability is another key attribute. Different techniques can be used to ensure the portability of monitoring systems, from using languages that are designed for portability, such as Java, to using network protocols to ensure interoperability with a large number of devices. Subramanyan *et al* [52] discuss their approach of using Simple Network Management Protocol (SNMP) to monitor heterogeneous devices in their network. SNMP is a network management protocol that defines a means of exchanging information between one or more network management systems and a number of agents within those systems. The protocol defines a means for formatting and storing management information [53] to enable communication and is widely supported amongst networked devices.

The collection of metrics is the basis of all hardware monitoring systems. Lucas *et al* [51] describe *black box* measurement through the collection “*of*

certain parameters, such as CPU and I/O time". NetLogger [54] is a monitoring solution that collects OS and Network level metrics using Unix utilities such as *netstat* and *vmstat*. The data structures provided by these utilities can include data such as CPU usage, virtual memory, disk activity and network transmission requests and can be harvested at various sampling intervals as the fidelity of the data requires.

It is not possible to collect all desired metrics directly. For those that are not collectable, they may be derived from a set of readings of other, collectable parameters. This is the basis for most modern hardware power measurement techniques that will be discussed in the next section.

2.2.2 Measuring energy consumption

The literature details three main ways of obtaining the energy consumption of computing equipment: direct measurement using specialized hardware, accessing system provided software APIs, or using a power models to estimate energy usage based on the relationship between the hardware utilization and energy usage.

Obtaining data about the energy consumed by a computer system is a task that has been traditionally achieved using dedicated hardware called Power Distribution Units (PDUs) to directly measure the electrical line supplying power to the system. PDUs are devices that distribute power to multiple electronic devices. In a data centre, they are generally rack-mounted and can be "dumb" if they do not provide instrumentation or management capabilities and "smart" or "metered" if they do.

Employing metered PDUs allows administrators to get an accurate view of how much energy their system is consuming, directly from the source. It is not influenced by the hardware architecture or software systems and can be further extended to instrument infrastructure such as network equipment, lighting and cooling to get a holistic view of how much energy the data centre is consuming.

Examples of the use of PDUs [55] or other similar devices like digital power analysers [56] to monitor the energy consumption of hardware devices can be

found in the literature. Pelley *et al* [57], have even developed a workload scheduling technique based on real time data about the power drawn from individual PDUs. They attempt to balance workload to evenly distribute the power load in a datacentre with the aim of avoiding overworking or stressing power delivery components.

Other approaches utilizing specialized hardware include the direct measurements of power lines on a motherboard [58][59], which has been proven to be accurate with an error range of around 5%. However, such an approach requires the cooperation of the motherboard manufacturer or a willingness to customize proprietary hardware that is not possible for most users who will treat their computers as black boxes. Even if such willingness was present, the intricate designs of a motherboard including the association between subcomponents and particular power lines may be considered proprietary by the manufactures and therefore not be in the public domain. Direct measurement by modification of the components may also lead to an undesired observer effect, which these papers do not discuss as a concern.

Fan *et al* [5] describe an approach used at Google to instrument a large number of machines in a data centre with physical power measuring devices. They attempt to correlate CPU utilization with actual power delivered to the entire system. Their aim was to find a curve that approximates this single activity level signal with the aggregate consumption of the system, something that would not be possible without the use of a PDU that provides information about the connected electrical appliance as a whole. The authors use only one metric to estimate energy usage over each machine. This differs from other work in the field, including the work presented in this thesis, that take into consideration multiple subcomponent activity levels to derive an estimation for power usage.

Fan *et al* instrumented a large data centre with thousands of PDUs to perform their comparison of energy consumption against CPU utilization. This would have been a costly and complex endeavour, a direct by-product of introducing a large number of new hardware devices into a building, and may be out of the reach of most organisations. Other approaches to direct measurement using specialized equipment, such as [58], [60] & [56] suffer from the same

lack of uneconomical scalability.

A different problem arises at the other end of the scale spectrum; if one wishes to obtain the power draw of a subcomponent of a machine or of a virtual machine, then direct power supply-based measurement is no longer sufficient, as noted by Stoess *et al* [61]. Direct measurement does not give insight into energy consumed by units smaller than a single physical server. A software approach is needed to obtain data on the energy consumption of these subcomponents or virtual devices.

To avoid the reliance upon hardware power measuring devices, software-based approaches have been developed to address the challenge of measuring energy usage accurately. Bertran *et al* [62] describe an approach where they access the performance monitoring counters (PMC) of a machine's power supply unit (PSU) to expose the energy usage information to monitoring applications that might want to use it. This type of software access to proprietary hardware is dependent upon the support of manufactures in exposing appropriate data, if available, and publicizing their API. Such access would also require PSU manufactures to agree on a common standard for accessing this energy consumption data or developers will be forced to write custom data collectors for each vendor.

HP, Intel, Microsoft, Phoenix and Toshiba launched ACPI in 1996 to consolidate power management functions and interactions between hardware and software into one open standard [63]. ACPI defines platform-independent interfaces for software services to access information about the hardware power status and modes and can therefore allow for the collection of hardware power metrics through software APIs. Yu *et al* [64] describe an example of using software agents to utilize Operating System calls to interface with ACPI-compliant hardware.

Such measurement APIs cannot match dedicated hardware for accuracy of power reporting as the specialized PDUs provide metrics for how much the entire electrical appliance is drawing, rather than what individual components believe they may be using. Misreporting of component usage will cause issues to the accuracy of the power data, and there may be components within the system that are drawing power but are not accounted for by a measurement

API.

An external monitoring mechanism should be able to provide a holistic view of the energy consumed compared to a self-reporting solution. Self-reporting solutions may be prone to miscalculations or errors when each subcomponent reports its own energy usage.

The widely available and accessible data on hardware resource usage leads to the possibility of an alternative strategy for collecting power data: estimation. It should be possible to collect hardware resource usage metrics and then use those metrics to indirectly estimate the energy consumption of the system.

Hardware resource usage metrics can be converted into energy consumption through the use of a power model. If a power model can be created that is sufficiently accurate, then it will be possible to estimate the energy usage of subcomponents and even virtual resources as long as they expose their hardware resource utilization levels.

2.2.3 Power estimation models

A power estimation model has to predict the energy consumption using hardware resource usage data. Most modern operating systems provide some means of access to the instrumentation of hardware subcomponents. The models described in the literature gather hardware resource data and transform it using a power estimation model. All of the models are devised after examining the relationship between the energy consumed and the systems resource usage, either through experimentation or simulation. Both of these approaches require applying statistical analysis to the data that has been collected [65].

How good the estimation will be is dependent upon the relevancy of the collected hardware resource metrics and the accuracy of the power model. If the model is dependent upon the system architecture or software workloads its usefulness may be limited.

In their paper, Kansal *et al* [9] note:

“In principle, VM power can be metered by tracking each hardware resource

used by a VM and converting the resource used to energy usage by way of a power model for those resources.”

The authors present a tool, *Joulemeter*¹², which uses only power models to accurately infer the energy consumption of a virtual machine.

The models are built, or trained, on servers with existing hardware power measurement capability and then are deployed across many of the same type of server without the need for additional hardware measurement. Their tool is proprietary and only executable on Microsoft systems. *Joulemeter* is designed to improve the estimation of electrical capacity planning for data centres and claims a 5% error rate. The authors have designed their tool so that it is more likely to over-estimate rather than under-estimate, an appropriate strategy given their aim of informing capacity planning. A data centre that was under provisioned electrically would result in servers without power to do work, where an over-provisioned server would only result in more electricity being available to use, at an increased cost to the organisation.

Kansal *et al* [9] used a commodity off-the-shelf power meter to compare their estimated results against power reporting from a dedicated hardware device. The device they use, the WattsUp Pro¹³, has an error rate of 1.5% plus 3 counts on the display value and does not guarantee billing-level accuracy for the power data. Using an off-the-shelf consumer device is strange in this situation as the tool is designed for a production datacentre. This would suggest that there is room for improvement by training a power model tool against more accurate metering hardware.

Fan *et al* [5] measured the total energy consumption of the computer hardware and found a strong correlation between the energy usage and the CPU utilization of the server. They state that the energy consumption of the server grows linearly as the CPU utilization increases, from an idle state where the minimum of energy usage is used to the maximum usage at full CPU utilization.

¹² <http://research.microsoft.com/en-us/projects/joulemeter/default.aspx>

¹³ <https://www.wattsupmeters.com/secure/products.php?pn=0&wai=0&spec=5>

Beloglazov *et al* [20] in their *Taxonomy and Survey of Energy-Efficient Data Centres and cloud computing systems* summarize Fan *et al* [5]'s model as:

$$P(u) = P_{idle} + (P_{busy} - P_{idle})(2u - u^r)$$

Where P_{idle} is the energy consumption when the server is not doing any work, P_{busy} is the power used when the CPU is full utilized, u is the CPU utilization and r is a calibration parameter. Using this model, the authors [5] claim an average error rate of <5% when tested across thousands of nodes under different types of workload. This model doesn't take into consideration the effect of other sub components. The author's explain this by suggesting high use of other sub components (e.g. I/O, memory) correlated with high CPU activity. There may exist situations where these other resources are heavily used and CPU is not. In such a situation, Fan *et al's* [5] approach would be invalid.

Meisner *et al* [41] found that on their test platform of a HP Blade Centre server, an idle server could consume up to 60% of its peak power when at 0% CPU load. This would give weight to the evidence that suggests energy usage is directly correlated with, or perhaps even linearly related to CPU utilization and other subcomponent use.

Bohra & Chaundary [6] develop the idea of using a power model to estimate energy consumption further by introducing additional terms to increase the sophistication of their model, which is aimed at predicting the energy usage of virtual machines. They have added metrics from the utilization of cache, RAM and hard-drives.

Their modelling technique, named *vMeter*, creates a linear weighted power model where the increased utilization of the subcomponents leads to higher power usage. Their proposed power model is based upon the linear relationship between the sub-components of the system and groups together hardware resources into CPU-bound (CPU, cache) and I/O-bound (RAM, HDD).

The authors decompose the total system energy consumption into two major categories: the baseline energy consumption (also known in the literature as

idle power, power that is consumed when the system is on but doing no useful work) and the dynamic energy consumption.

Baseline power is given by (notation is preserved from Bohra & Chaundary's [6] work)

$$P_{baseline} = \alpha * a_1 + \beta * a_4$$

Total energy consumption is given by:

$$P_{total} = P_{baseline} \sum_1^k P_{domain(k)}$$

Domain power:

$$P_{domain(i)} = \alpha (a_2 * P_{CPU(i)} + a_3 * P_{cache(i)}) + \beta * (a_5 * P_{DRAM(i)} + a_6 * P_{HDD(i)})$$

Key:

Symbol	Meaning
P_x	Power with respect to domain x
a_n	A coefficient or weight variable for the paired power domain
α	CPU + Cache power intercept
β	DRAM and HDD power intercept

The model sensibly breaks down energy consumption per domain to the power calculated for each resource multiplied by some weight to achieve the total power for the system. However, weights $\alpha, \beta, a_1, a_2, a_3, a_4, a_5, a_6$ are workload specific and are calculated based on each benchmark program. Furthermore, this calculation appears to be done manually in preparation of each benchmark, an approach that will not scale when such a system is

deployed in a large data centre.

Using this method they claim 94% average accuracy when compared against the actual measured power using an externally attached power-measuring device. Their configuration was tested on an unspecified number of AMD Opteron Sun Sparc servers. The accuracy of the power meter they used might be questioned despite the authors claiming it is “*inexpensive but sufficiently accurate*”, as it is the same consumer off-the-shelf device as also used by Kansal *et al* [9] and does not guarantee the billing-level accuracy that is normally provided by a PDU in data centre environments.

Chen *et al* [7] ratify the results of Bohra & Chaundary [6], but attempt to simplify them by removing the memory component. They found the energy consumption of the memory to be ~5W, which they consider to be negligible since it accounts for only ~5.5% of the idle energy consumption (90W) or ~3.5% of the maximum energy consumption of (140W). This development could be questioned; because, while memory energy consumption may be small, it is still significant in terms of gaining an accurate estimation of the energy consumed. Neither Chen *et al* nor Bohra & Chaundary consider the impact of network activity in their work. Chen *et al* do not disclose the accuracy of their model and described their System Under Test (SUT) as one node featuring an Intel E560 chip.

Chen *et al* [8] also divide energy consumption in two parts: 1) fixed energy consumption (idle time) and 2) variable energy consumption (also known as dynamic energy consumption in the literature and here means the energy consumed by tasks). They introduce the concept of a task as a logical unit, for which energy must be expended. Their rationale is that a significant impact on energy consumption and system performance can be experienced, dependent on the workload and type of task. They compute the energy consumption of a system as the sum of all tasks that system is computing, but they did not investigate the consumption of mixed workload tasks so their results are limited. The accuracy of the authors’ proposed model is not discussed in their evaluation. They do not provide details of their test bed, except to describe their power meter from which they calibrated their model, which was again a consumer off-the-shelf device, designed for home use (GreenWave

PowerNode¹⁴).

A task-based approach to estimating energy usage is interesting, as the amount and type of work a computing system does has the single largest influence over its dynamic power usage. However the main challenge remains the accurate assumption of how much energy each type of task requires and will result in the model requiring training for each type of task, an endeavour that may require constant computation as new tasks are assigned.

Table 2.1 summarises the different power models in the current literature, showing error rates, resources used in the models, test bed facilities and hardware power measuring devices that were used to calibrate each of the models.

2.2.4 Summary

In summary, power models provide a good method for estimating the energy consumed by a server or virtual machine and have been demonstrated to be accurate with an error rate of up to 6%.

The scalability of such power models is one of their key advantages and helps to explain their suitability to cloud computing environments. However, they are not without disadvantages: the accuracy of a power model depends upon its fit to the hardware and the workload that is being executed, the literature includes models with varying degrees of accuracy which depend on the specialized hardware power meters that they are trained with. A low quality power meter will influence the model and reduce its effectiveness.

¹⁴ http://greenwavereality.com/downloads/pdfs/PowerNode_manual_english.pdf

Authors	Resources Used	Est. Error	Test facility	Power Meter
Fan [5]	CPU	<1%	Google Datacentre (thousands of nodes)	Data centre PDU, model or accuracy not discussed.
Bohra [6]	CPU, cache, HDD, RAM	6% average	AMD Opteron Sun Sparc Servers. Undefined number.	WattsUp Pro. +/- 1.5% plus "3 counts on the display".
Chen [7]	CPU, cache, HDD	Not given	1x node with Intel E560 CPU	Schleifenbauer PDU (model or accuracy not given)
Chen [8]	Task as logical unit	Not given	Not provided	GreenWave PowerNode
Kansal [9]	CPU, Memory, Disk	5%	1x Dell PowerEdge R610	WattsUp Pro. +/- 1.5% plus "3 counts on the display".

Table 2.1: Comparative table of power models

Any power model must be computed, a process that will consume computing resources. To avoid or minimize the observer effect, any software based power estimation should have a minimal footprint. If the power estimation metrics will be used to influence workload allocation or other system behaviours the data must also be timely, so the latency of the computation must also be considered.

The challenges of an effective power model are to be accurate, scalable and with low overhead or computing load. The software based energy measurement tool that is presented in this thesis aims to address some of these challenges and is described in depth in Chapter 3.

2.3 Workload mixing

2.3.1 Approaches to workload allocation to manage energy

Organisations that require server capacity do not want to be concerned with virtual machine management. They want to run their applications and it is left

to system administrators to provide them with that capability at the lowest cost. One approach to do this is to use virtualization technology to consolidate applications. It allows multiple servers to be packed onto a single physical host, dividing the physical resources amongst each virtual machine. Without virtualization, each user would require their own dedicated individual server, increasing the number of physical machines required [66].

Virtualization technology has a performance penalty when compared with running applications of the native OS. Figures between 10-15% reduction in throughput have been reported [66][67] and in some cases a 20-60% increase in response times. The performance impact must be managed as it has been shown to be dependent upon the load of the host, with highly loaded machines suffering degraded application performance.

However, McDougall and Anderson [68] note, *“Over the last five years, the throughput of a virtualized host as measured with VMmark has doubled every year.”* Thanks to improvements in the virtualization software stack and in hardware architectures it seems as if the use of modern virtualization technology may not have significant performance penalties. If there are performance penalties, those penalties may not outweigh the benefits of consolidation, multi-tenancy and dynamic provisioning that virtualization can provide.

Application consolidation using virtualization is a widely used strategy to increase the energy efficiency of computing systems. El Rheddane *et al* note [69]:

“Consolidation is a well known solution for energy saving in the context of virtualized systems... The placement policy takes into account the CPU and memory usages, in order to concentrate the VMs on fewer nodes of the datacentre, thus allowing unused nodes to be shut down, or put into low-energy mode.”

Chen *et al* [35] concur:

“Server consolidation is a powerful tool which has been widely adopted to gain high energy efficiency of server, which results from keeping active servers in high utilization by turning off overprovisioned servers.” [70][71]

Srikantaiah *et al* [10] note that such consolidation requires a balance between energy savings and higher contention rates. The goal is to keep servers well-utilized so that power costs are effectively amortized. This is balanced against over-utilization that can cause internal contentions such as cache contentions, conflicts at the functional units of the CPU, disk scheduling conflicts, and disk write buffer conflicts.

Srikantaiah *et al* model their approach to loading servers to a desired utilization level as a multi-dimensional bin packing problem where servers are bins with each resource (CPU, disk, network, etc.) being one dimension of the bin. The bin size along each dimension is given by the energy-optimal utilization level. They note the need to balance consolidation strategies with application performance.

Another approach to highly utilized servers by Chen *et al* [11] points out that it may be possible to rewrite load balancing algorithms to be more energy aware and introduce the concept of “*load-skewing*” as opposed to the more traditional approach of “*load-balancing*”. In this context, *skewing* can be taken to be a workload consolidation approach. *Load skewing* aims to continue to consolidate applications on physical servers as long as there are resources available to do so. This will minimize the number of machines that are required to do any given set of work and thus allow unused machines to be powered down.

When allocating work to physical hosts, the allocation is normally done at once when the system receives the request for work, however later in time the system workload can be reorganized and re-allocated using live migration of virtual machines. This allows the system to be continually adapted to the changing workload conditions and always be in a state of most efficient workload allocation.

However, a live migration strategy can be costly. Dynamic reconfigurations of workload may be subjected to switching costs in terms of the energy consumed while a machine is being turned on/off. Srikantaiah *et al* note that this cost may sometimes overshoot the benefit of the reorganization [10] and Petrucci *et al* [72] found the cost to be so significant that they introduced a penalty value in their configuration model to estimate the cost of this live

migration. Live migration can also cause a negative impact on system performance such that SLA's are violated, particularly in high-availability systems [73].

Alternative strategies have emerged that aim to consolidate applications while being considerate of application performance. These attempt to optimize for both energy and performance.

One approach is to focus on improving the performance of applications that are traditionally hit hardest by consolidation, such as I/O bound tasks. Kim *et al* [12] present a task-aware virtual machine scheduling mechanism that improves the performance and responsiveness of I/O bound tasks within VMs, which are normally most under threat from contention with CPU bound tasks for CPU fairness. I/O bound tasks are identified in mixed workload situations and are *boosted* in scheduler priority to gain improved access to resources.

An alternate approach by Moreno *et al* [13] aims to reduce the phenomenon of *Performance Interference* that arises when the resource consumption of a VM impacts another VM running on the same server [74].

Moreno *et al's* [13] approach is to schedule away from this interference to improve energy-efficiency. If jobs are seen to be impacting those around it, they are moved onto the unused servers. This solution is based upon data from Google cloud's tracelog, data that reports the utilisation of an extremely large number of Google servers over a long timespan. This data provides information about CPU and MEM usage for that large number of machines. The authors simulate their approach to interference-aware scheduling on CloudSim, a popular open-source simulator of cloud environments.

There are two main differences between Moreno *et al's* work and the work presented in this thesis. Firstly, Moreno *et al* schedule away work using live migration when interference is detected. The placement of VMs during this migration is irrespective of the type of work performed or if the new placement will have a better effect on application performance. If a new location is found to be unsuitable then the VM is transferred again once the performance interference metric passes a threshold value.

Secondly, Moreno *et al's* approach models a data centre with “*highly heterogeneous resources*”, a characteristic they exploit to select the most appropriate hosting location for each task. This approach might not be appropriate to apply in all situations, as some data centres will be made up of standard, homogeneous, off-the-shelf servers.

Table 2.2 summarizes each of the approaches in literature to VM allocation strategies for energy saving. There are four main approaches: bin packing, load skewing, boosting specific tasks and interference aware scheduling. 50% of the approaches use live migration in their solution. 75% report reductions in energy usage and a different 75% also report significant application performance degradation.

Authors	Approach	Live Migration	Energy Savings	App. Performance impact
Srikantiah [10]	Bin-packing (consolidation)	Yes	Algorithm designed to minimize energy subject to performance constraint. Constraint is modifiable	Application performance is degraded, but a limit is placed on the degradation. That limit is not specified in the evaluation.
Chen [11]	Load-skewing (consolidation)	Undefined	Between 20.2% – 30.8% depending on algorithm and parameters.	Dependent on algorithm parameters. Cost of service degradation can be high, with the highest energy savings reported millions more Server-initiated disconnects than the baseline.
Kim [12]	Boost I/O intensive tasks	No	Not provided.	I/O response time drops from ~70ms to ~5ms, a 13x in I/O responsiveness.
Moreno [13]	Interference-aware scheduling	Yes	15% improvement in datacentre efficiency	27.5% reduction in VM interference.

Table 2.2: Comparative table of VM placement strategies

Our work, described in Chapter 4, describes an approach to VM placement on a homogenous private cloud system where tasks are considered with respect to resource usage of four subcomponents, CPU, Memory, Hard Disk and Network. A key assumption in our approach is that under-utilized machines are not powered down, a point that most energy efficient algorithms ignore.

Traditional research in this area has focussed on reducing the number of physical machines required so that under utilized machines can be powered down, something which we believe is impractical in most data centres. As we have seen, idle servers can consume up to 60% of their peak energy usage [41], so if those machines are not powered down then a different approach is

needed.

In the paper “*Failure Trends in a Large Disk Population*”, written at Google, Pinherio *et al* [75] write:

“As is common in server-class deployments, the disks were powered on, spinning, and generally in service for essentially all of their recorded life.”

This insight would suggest that it is common in data centres for all machines to be on and in service essentially all of the time, and are not powered down for energy efficient reasons. This could be because hardware lifespan can be impacted by heavy server utilization [35] and an increased number of power cycles [75].

Barroso and Holzle [22] comment that powering down components “*complicates application deployment and greatly reduces their practicality*”. The authors argue that because data is distributed as well as applications, powering down servers may require costly and time-consuming redistribution of this data if system dependability were not to be compromised, software managing these systems would also increase in complexity.

It is also likely that servers are not powered down simply for application performance reasons. The speed at which work can be done on computers is generally the primary concern of most organizations; it’s what drives purchase decisions for new hardware. It is always desirable to be able to do work faster or be able to do more work at once. Over-utilizing machines will degrade application performance, impact system responsiveness and violate SLA’s [39]. In such situations, energy usage is likely to be a secondary concern. Barroso *et al* [22] note that components in a sleep state have “*high wake-up penalties*” that will impact system responsiveness, so resources are unlikely to be transitioned to low power states

Such a reality may impact the adoption of hybrid datacentres [76] that mix high performance servers with dedicated low-power machines. The authors found that the higher performance servers delivered nearly twice the performance of the low power machines, which suggests that using low powered machines is only appropriate for low-resource usage workloads.

Such scheduling requires identification of lightweight jobs and a system scheduler with knowledge of the underlying hardware capabilities to map the appropriate low intensity jobs to energy-efficient hardware.

Latency-sensitive tasks have sometimes simply not been allowed to be co-located with other tasks because the performance degradation means that SLA's can't be guaranteed. Attempts have been made to improve the predictability of degraded performance in order to allow time-sensitive tasks to be co-located and therefore gain the benefits of higher utilization without breaking SLA's because of unknown performance degradation. If the performance hit is known, then an appropriate SLA can be declared [77].

Given then the demand for high performance of our applications, is it possible to arrange workload in such a way that performance is maintained but the organization energy bill is reduced? Srikantaiah *et al* [10] note, "*...there has been little work on joint power and performance aware schemes for multi-dimensional resource allocation*". That is the challenge that is addressed and described in Chapters 4 and 5 of this thesis.

2.4 Modifying OpenStack

This section covers the relevant academic literature on modifying private cloud systems to improve energy usage or application performance. The current literature includes attempts to implement workload allocation strategies on OpenStack or other private cloud systems. There is not much literature in the modification of OpenStack, so work on the modification of the Eucalyptus private cloud system is also considered.

NASA and Rackspace Hosting founded the OpenStack cloud computing project in July 2010 in a joint venture¹⁵. The project is intended as an *Infrastructure-as-a-Service* platform, allowing organizations and individuals to deploy cloud computing services without specialized hardware.

¹⁵ <http://www.rackspace.com/cloud/OpenStack/>

OpenStack has released seven versions since its inception and between the most recent Diablo and Essex versions a large change in the architecture of the scheduler was seen. Essex brought in the adaption of the *FilterScheduler* architecture that is discussed in more detail in Chapter 5.

2.4.1 Modifying private cloud scheduler for performance or energy purposes

Continuing on from the workload allocation strategies that were discussed in Section 2.2.2, Liao *et al* [14] describe adapting an SLA-sensitive energy efficient allocation strategy that has been evaluated on the OpenStack private cloud platform.

Their placement strategy attempts to allocate VMs on the lowest number of physical hosts, but their aim of non-violation of SLAs contrasts with other approaches that aim to minimize the amount of energy consumed. During discussion of their experimental results the authors note that their strategy is sufficient to *meet all* SLAs, but don't describe any other aspects of application performance. It is sufficient that the applications only perform as well as required to meet their contracts.

The authors estimate power as a function of CPU utilization, which, as discussed in Section 2.1 of this chapter, may not give an accurate estimation of the amount of energy consumed. The experimental results show that the author's system does indeed require less energy than the standard OpenStack placement strategies of *random* or *round robin* without violation of the defined Service Level Agreements, but they do not describe how they developed the integration of their algorithm with the private cloud system.

Corradi *et al* [15] also present an approach that focuses on consolidation to save energy. Their paper notes how it is important to carefully consider the degradation of performance that may occur due to resource contention between co-located virtual machines, but they do not describe a solution for addressing this issue.

The authors provide evidence that CPU load and energy consumption increases as the number of machines consolidated on a single physical host

increases. This has a detrimental effect on application performance such that user response times increase, as does the number of requests that fail. The consolidation system they describe has been implemented on an OpenStack private cloud by they do not detail how the implementation was achieved.

Beloglazov *et al* [16] propose a framework for optimizing OpenStack to improve energy efficiency. They outline in a blueprint document (date 14th August 2012) how such a system could be developed and their intended strategies for workload allocation. The current status of the work is not known at the time of writing as no new documents have appeared on the Internet from these authors since August 2012.

Beloglazov's project "OpenStack Neat" [16] aims to provide a dynamic consolidation strategy to OpenStack to improve machine utilization and reduce energy consumption. Live Migration is to be employed to dynamically re-allocate machines in response to real-time resource demands. Idle machines will be moved to a sleep mode.

In their proposed architecture, each physical host is given a "Local Manager" alongside the OpenStack *nova-compute* daemon. The *Local Manager* monitors the resource usage of the physical host in real-time and triggers migration of its virtual machines when one of two conditions occurs: *Overloaded* is defined by the authors as when a host is highly utilized such that applications within the guest VMs may be experiencing performance degradation. *Underloaded* is the opposite condition, where a physical host has running VMs that could be placed elsewhere allowing this machine to be switched to a low powered mode, such as sleep mode.

The blueprint for this system is sound, but as it is incomplete it is difficult to evaluate in the context of our work. The virtual machine allocation strategy of a consolidation-focused approach to saving energy is similar to those described in other parts of the literature, as discussed earlier in this chapter.

There does not yet seem to be an available update to the document that details finished work, and there was no activity on the project's *github.com* repository

between February and September 2013 ¹⁶.

In a Master's thesis submitted in July 2013, Lindgren [17] describes development of a hybrid solution to optimal VM placement for OpenStack that consists of an initial placement decision (the focus of the 2013 thesis [17]) and a dynamic live migration system (which is the focus of an accompanying 2012 research paper [78]). The Lindgren thesis focuses on the engineering effort to develop the experimental system.

This work is currently the only academic work that goes into a degree of detail as to how placement allocation strategies are developed for the OpenStack private cloud system. The scheduling algorithms are implemented as *cost functions*, modified from the original scheduling mechanism in the *Diablo* release of OpenStack. Our work is based upon a newer release, *Essex*, and as such is developed around the *FilterScheduler* architecture that is discussed in Chapter 5. *FilterScheduler* changed the way VMs were allocated on OpenStack, introducing a more sophisticated two-step comparison process in-place of a simple cost function. Therefore it would be inappropriate to directly compare the results of Lindgren's allocation strategy with that presented in this work.

Lindgren develops two simple VM scheduling strategies that are examples of *stack* (consolidated) and *spread* (balanced) approaches to workload allocation. The strategies are evaluated in experiments to show that consolidation will rarely exceed the theoretical minimal number of servers required and spread will ensure that maximum utilization of physical hosts does not stray far from the average. Data for application performance or the amount of power used by the physical hosts are not given. All workload is generated using a synthetic workload generator that requests and terminates VMs on a continuing basis through the length of the experiment.

¹⁶ <https://github.com/beloglazov/OpenStack-neat>

Authors	Approach	Application Performance	Energy Savings	OpenStack Version
Liao [14]	Consolidate with respect to SLAs	Only sufficient to meet SLAs	Significant, but figures not provided and unable to read from graph	Not Provided
Corradi [15]	Consolidation	Not provided	Very basic analysis, based on consolidation. Example given is if VMs from 5 machines can be consolidated to 1 then 80% reduction in energy is achieved.	Diablo
Beloglazov [16]	Live Migration consolidation	No evaluation	No evaluation	Not provided
Lindgren [17][78]	Hybrid initial placement/live migration system. Either consolidated or balanced	CPU utilization maximum was never more than 10% from the average, but no figures given of application performance.	Limited to noting that only one more host was than would be required in an optimal layout.	Diablo

Table 2.3: Comparative table of approaches to modifying OpenStack

Table 2.3 shows a summary of the different approaches to modifying the VM allocation strategy on OpenStack. All four approaches are predominantly concerned with consolidation of VMs, two of which purport to use live migration although one of those remains unimplemented. All four pieces of literature do not report application performance statistics, although in one case SLA violations are included. These references also lack proper analysis of their energy saving claims. If analysis is present, and in some cases there is no analysis at all, there is no published data on usage other than the number of machines used. Those that do present data only reference the number of machines used as a metric from how much energy saved. They do so simplistically, claiming that if 5 hosts worth of VMs was consolidated on to 1 machine then and 80% reduction in energy usage was achieved.

Aside from OpenStack, there has been work on virtual machine allocation to reduce energy consumption on Eucalyptus, an alternative open source private cloud infrastructure system: Graubner *et al* [79] present a live migration system that attempts to reorganize the current VM allocation into a more efficient state. Their work doesn't take into consideration the nature of the applications within the VMs being migrated and acknowledge that in some cases, migration may lead to situations where application performance is further degraded due to inappropriate coupling of certain VM types. Power savings are limited to reducing the number of physical machines that are in use and switching unused machines to low-power states.

Based on the current state of the literature in this area, there is scope for development of a virtual machine allocation system on OpenStack that limits the amount of energy consumed while attempting to preserve the performance of applications. Current work focuses on the gains in energy savings that can be made by reducing the number of physical machines that are required, but as we have seen in Section 2.2, the shutting down of unused machines is not always possible or desirable. The development of a system that fits these requirements using appropriate coupling of virtual machines is discussed in Chapter 5 of this thesis.

2.5 Research methods

During the initial stage of our research we attempted to understand the issues that organizations face when trying to reduce their energy costs. Once it was determined that software based energy measurement might be an appropriate strategy to pursue, empirical investigations [80] took place to guide the development of an appropriate tool.

Observation of the computing system and resource usage helped to develop models for energy usage by observing and identifying the relationships between resource usage and energy consumption, such a process is common in the field of Systems Development Research. [81]. Modelling the observed relationships, as detailed in [82], helped to improve our understanding of the computing system and led to the development of energy estimation software.

The workload mix investigations were similar in nature, where exploratory experiments took place to attempt to understand the trade-offs between energy usage and performance in private cloud systems. Feedback from these experiments helped to develop new models for experimentation and focused the research on workload mixes that could actually improve the characteristics of our system, a process of experimentation that is well understood as the scientific method. [83].

It is expected that an experiment will normally have a *null hypothesis* that states that there will be no difference between the experiment results and an alternative hypothesis that cannot be true if the *null hypothesis* holds [80][84], this is an example of Hypothesis testing; another well understood approach to experimental research [85]. We followed this approach to develop our workload mixing strategies, by stating *null hypotheses* that were then refuted these using statistical evidence. The significance of the statistics produced by our experiments was analysed using T-Tests to determine how confident we should be about our empirical results. Our use of this strategy, including the statistical analysis of our experiments is discussed in Chapter 4 and 6.

Finally, the development of a workload placement allocation engine for OpenStack was evaluated and verified by comparing its allocation strategies and their impact on performance of applications and energy usage with those generated by alternative workload placement (described in Chapter 5 and 6).

The allocation strategy for OpenStack was evaluated with a synthetically generated workload, for which there is precedent in the field [7][86][87]. Synthetic workload generation is designed to provide a stable and predictable workload for evaluating systems. It allows the developer to specify conditions that he wishes to evaluate and helps to provide reproducibility in the results.

3 Software based energy measurement

3.1 Introduction

Based on historical data, energy costs are rising at 15% per annum in the United Kingdom. The rate of energy inflation is higher than the rate of inflation of other infrastructure costs, such network access, so that electricity costs are becoming a larger percentage of overall infrastructure cost. To help understand and manage these costs, we need to monitor the energy usage of data centres, with readings of the power drawn in real time and cumulative lifetime totals required.

However, an NCC UK report estimated that only 13.4% of organizations monitor energy consumption of any kind [4]. Introducing monitoring of energy consumption will not only inform users and administrators about their consumption levels, but will also have the potential to impact how computing hardware systems are operated by enabling management policies that adapt to energy usage.

Power usage data for computing hardware is normally obtained by one of three ways:

- 1) Polling Power Distribution Units (PDUs) connected to IT equipment.
- 2) Collecting metrics from hardware performance counters
- 3) Estimating usage using metrics for other resources such as CPU.

PDUs monitor the physical lines providing electricity to the computing hardware so can provide accurate data about the consumption of the whole machine. However, each additional PDU comes with an expenditure cost and will require additional maintenance. If the number of machines is increased, a PDU is required for each new machine, so that using PDUs at data centre scales is very expensive.

To avoid the reliance upon hardware power measuring devices, software-

based approaches have been developed to address the challenge of measuring energy usage accurately. Software access to performance counters provide the data required. This type of software access to proprietary hardware is dependent upon the support of manufactures in exposing appropriate data, if available, and publicizing their API. These so called “self-reporting mechanisms” may also be prone to miscalculations or errors when each sub component reports its own energy usage.

The third way of collecting performance metrics is to use a power estimation model to predict the energy consumption using hardware resource usage data. Power models are devised after examining the relationship between the energy consumed and the systems resource usage, either through experimentation or simulation. How good the estimation will be is dependent upon the relevancy of the collected hardware resource metrics and the accuracy of the power model.

To allow organisations to accurately collect power measurements, *CloudMonitor*, an open-source, automated, scalable, resource and power-usage reporting tool has been developed. *CloudMonitor* can provide fine-grain utilization data from the physical hardware subcomponents and generate power models that map the use of these resources to the overall machine power usage. A power model is a mathematical expression that attempts to capture the relationship between subcomponent use (such as CPU, DISK, etc.) and energy consumed by the machine. These models can then be used to provide software-based power estimation in both virtualized and physical computing environments at scale.

CloudMonitor is one of the major contributions to this thesis and was designed to help organizations understand their energy usage as a basis for decision makers to take steps that reduce their energy consumption. *CloudMonitor* may be deployed on any machine, physical or virtual, that is to be observed. It will then interact with the host operating system to gather metrics on resource usage and if configured to do so, estimate power usage.

The tool’s power model generation facility provides software-based energy measurements based on subcomponent resource usage. This facility provides an accurate estimation of energy consumption that can be scaled without

requiring additional power metering hardware.

CloudMonitor also opens up the possibility of an energy tariff for cloud computing systems. If the financial cost of energy required for virtualized resource can be calculated then those costs could be used for cloud providers to bill their users according to energy consumed. Section 3.4.3 of this chapter details such a potential tariff based on the resource usage of the *CloudMonitor* evaluation experiment.

This chapter describes the design, implementation and evaluation of *CloudMonitor*. This chapter starts with a discussion of the power model generation (Section 3.2), then moves on to describe the design of *CloudMonitor* (Section 3.3), including the sub-modules, storage architecture and data collection. The final section (3.4) describes the evaluation of *CloudMonitor* monitoring a typical web application that has multiple VMs in its deployment.

3.2 Power model-based estimation

CloudMonitor provides the capability to estimate energy usage through a software-only model that is initially “trained” using real-time energy usage information from PDUs. The training phase allows *CloudMonitor* to ascertain the connections between resource consumption and power used to generate a mathematical model that can be then rolled out across the system for all machines of the same configuration.

To automate the process, we use the Multiple Linear Regression class of *Michael Thomas Flanagan’s Java Scientific Library*¹⁷ to accurately assess the coefficients of the independent variables

This approach is developed from work on power models in the literature, discussed in Section 2.2.3 of Chapter 2 of this thesis, in particular, *vMeter* by Bohra & Chaundary [6]. The authors describe a method for predicting energy

¹⁷ <http://www.ee.ucl.ac.uk/~mflanaga/java/>

usage of virtual machines by monitoring consumption of hardware resources on the host server - in particular CPU, cache, RAM and hard-drive. Their proposed power model is based upon the linear relationship between the sub-components of the system.

However, the weights in the power model of *VMeter* [6] are workload specific and are calculated manually for each individual application. *CloudMonitor* automatically analyses resource consumption to create models that are applicable for the current server configuration under any workload.

vMeter's power weights are also calculated by calibrating the power model against an off-the-shelf consumer level hardware power measuring device. The accuracy of this device is questionable as it does not guarantee billing level-accuracy. In my work the power model for *CloudMonitor* was calibrated against a professional data centre PDU. Each server was connected to a socket on the Rairitain PX-5367 PDU¹⁸ allowing real time billing level monitoring of power usage. The Rairitain PDU provides per socket (effectively per server) energy usage information including wattage, current and voltage drawn, to give an accurate picture of energy consumed by each physical server.

For a batch of machines of the same type and configuration, training of the model is only required on one machine per batch. The resulting model is able to predict energy usage across the remaining servers of that type without the need for additional dedicated metering hardware. If the hardware and software (including Operating System and Drivers) configuration is the same across multiple machines then the power model is applicable even for different workloads.

Our power model is:

$$Power = \alpha + (\beta_1 * CPU) + (\beta_2 * Memory) + (\beta_3 * Hard Disk) + (\beta_4 * Network)$$

This model takes into account each hardware resource that we measure and

¹⁸ <http://www.raritan.co.uk/downloads/datasheets/dominion-px>

generates the weights α and $\beta_1, \beta_2, \beta_3, \beta_4$ automatically during the training phase. *Power* is the current active power as calculated by the model. *CPU* is the current CPU utilization as a percentage. *Memory* is the current amount of memory in use, expressed as megabytes. *Hard Disk* is the number of bytes written and read since the last sample. Similarly, *Network* is the number of bytes sent or received by the machine through its network interface since the last sample. α and β_n are coefficients, generated automatically during the model training. α is approximately equal to the baseline energy usage when the system is idle.

Some models in the field [5] are non-linear in their expression of how resource consumption effects power usage, particularly in relation to CPU utilization. The model presented in this work found that an accurate result could be achieved with a linear model as shown in the evaluation experiment later in this chapter.

The sample time for the *CloudMonitor* implementation of this power model is every three seconds, as discussed on page 57. *CloudMonitor* design

3.2.1 Architecture

Analysis of the literature in this area leads to three objectives that the *CloudMonitor* tool should meet: it must be scalable, collect all of the required data and estimate energy usage accurately. The latter is achieved through the use of the power model discussed in Section 3.1. This Section details how a scalable software architecture to collect all of the required data dynamically is achieved.

To be scalable, *CloudMonitor* must adopt an architecture that makes it suitable for an environment with a large number of machines. Yu *et al* [64] discuss a number of possible architectures suitable for collecting and analysing monitoring data in a cloud-computing environment with a large number of machines. We broadly follow their recommendations with *CloudMonitor*; adopting a structured monitoring platform and a set of data-stores to maintain the information collected by agents. The agent numbers are unlimited, and report their information back to their assigned data store, thereby reducing

and partitioning the load as required. The architecture of our data storage is described in Section 3.3.6.

A diagram illustrating the architecture of *CloudMonitor* can be seen in Figure 3.1.

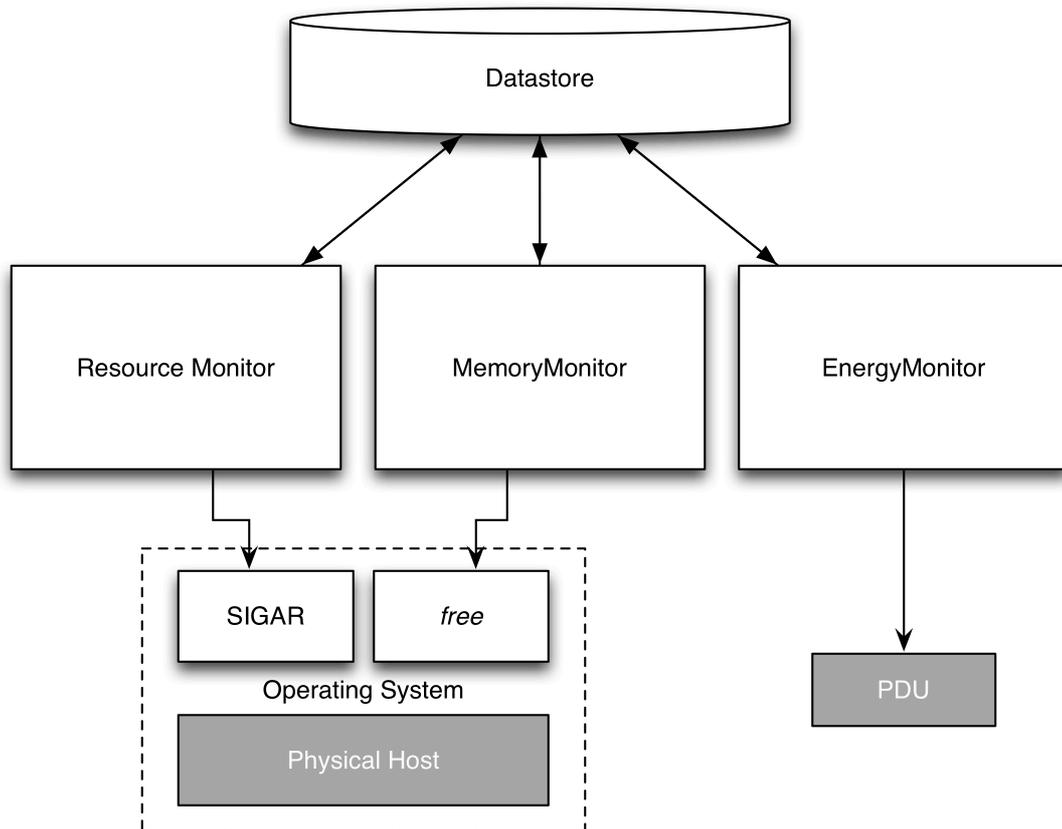


Figure 3.1: *CloudMonitor* architecture diagram

There are three modules that handle the *CloudMonitor* software data collection, *ResourceMonitor*, *EnergyMonitor* and *MemoryMonitor*. They each follow the same workflow operation that is illustrated in Figure 3.2. Each of the tools is responsible for collecting data from their assigned metrics: *ResourceMonitor* gathers information on system subcomponent usage from the OS using the *SIGAR* library, *EnergyMonitor* uses SNMP to poll any connected hardware power measuring devices and *MemoryMonitor* polls the OS to ask for memory usage statistics. Each of these modules is explained in more detail in individual sections below.

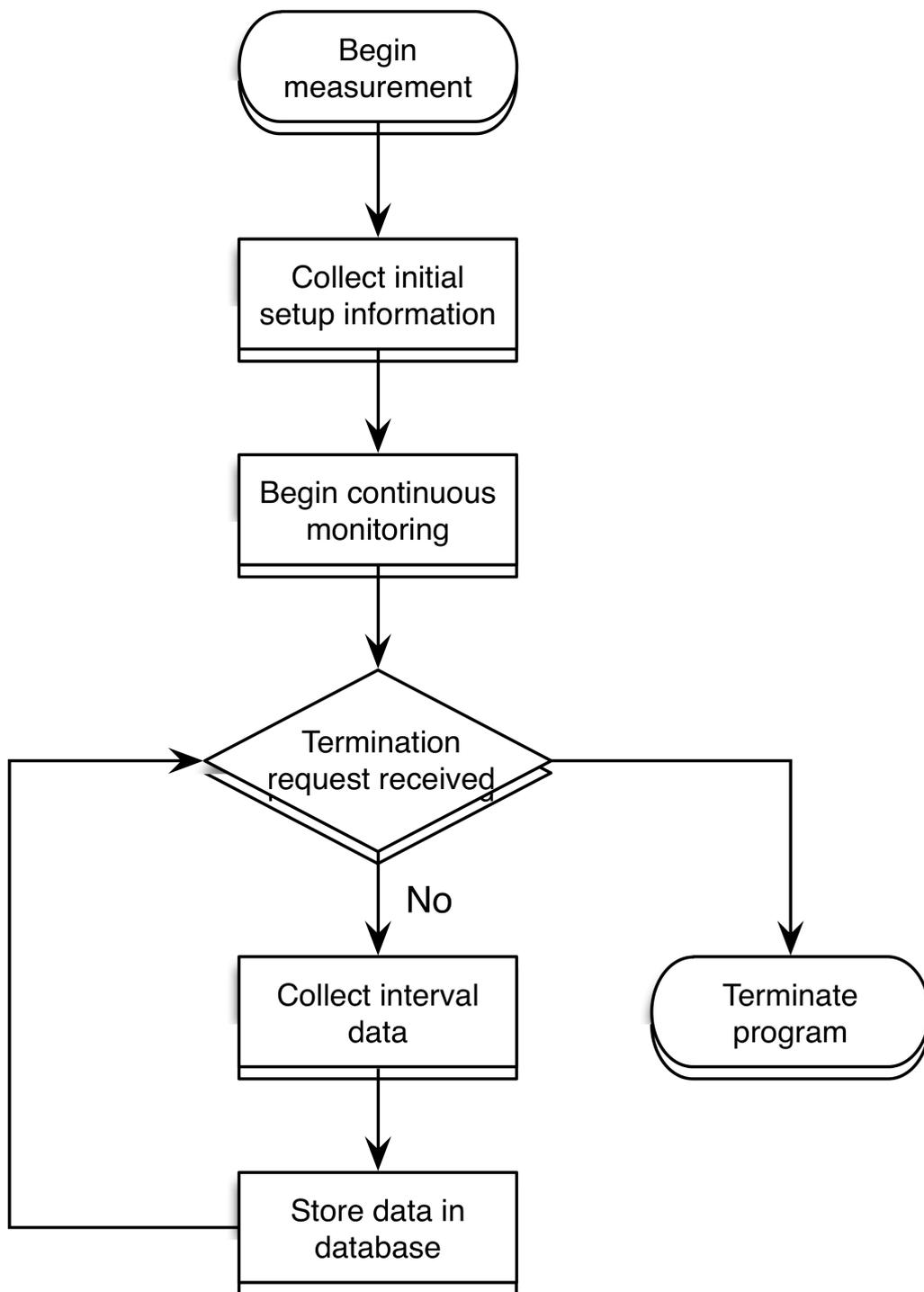


Figure 3.2: *CloudMonitor* continuous operation workflow

The software in the tool was developed using the agile method *Scrum* over an 8 month development period, with regular sprints followed by demonstrations of emerging functionality. *CloudMonitor* and assorted tools consist of around 3,000 lines of program code, mostly Java, and are available

online under the open-source Apache license. The code is available on Github.com¹⁹, which was used during development along with the *git* source code management and revision tool to maintain our project files.

3.2.2 ResourceMonitor

The *ResourceMonitor* module collects usage information about the system sub-components (CPU, RAM, HDD, Network) using Hyperic’s “System Information Gatherer” (*SIGAR*) Library²⁰. The metrics collected are detailed in Section 3.3.5.

SIGAR provides a cross-platform API for collecting data on sub-component resources and application processes by reading hardware performance counters. Each platform that is compatible with *SIGAR* has a native library that interfaces with the local operating system to request metrics such as current CPU utilization, hard disk operations, etc. The choice of this library was an implementation decision based on our preference to use the Java programming language. Similar functionality could have been achieved by recording system events [88], or by accessing hardware performance counters [45]. This would have required programming a custom tool using low-level system calls and languages, so it was decided that using the existing library to provide this functionality would be good Software Engineering practice.

```
MachineUtilisationData results = new MachineUtilisationData(machineID);
try {
    CpuPerc cpu = this.SIGAR.getCpuPerc();
    Mem mem = this.SIGAR.getMem();
    FileSystem[] fslist = SIGAR.getFileSystemList();
}
```

Snippet 3.1: Example of *SIGAR* calls to gather resource usage metrics

¹⁹ <http://github.com/jws7/cloudmonitor>

²⁰ <http://www.hyperic.com/products/SIGAR>

The initial Java code of *ResourceMonitor* was forked from the automatic monitoring software of *H2O*, a distributed database system developed by Angus Macdonald at St Andrews [89], which in turn was based on the monitoring system *NUMONIC* developed by Graham Kirby, also at St Andrews [90]. To this code we have added the original *MemoryMonitor* and *EnergyMonitor* modules and have rewritten the *ResourceMonitor* to collect, process and store the information required for our purposes.

An example of the data collected at each sample interval are detailed in Table 3.1:

Date	2013-08-28 22:43:03
Machine ID	DE:3E:D5:9B:22:CA
CPU used	0.0012
CPU idle	0.9988
Fs_disk_reads_bytes	1266603008
Fs_disk_writes_bytes	5864076
IP	138.251.198.20
rx_bytes	6990072976
tx_bytes	180923493

Table 3.1: Example of *ResourceMonitor* collected data.

CloudMonitor can also collect power data to match the machine resource usage information. If the tool is configured to collect PDU data and a map is provided to match the machine identifier to a PDU socket, the *EnergyMonitor* component is invoked.

3.2.3 *EnergyMonitor*

EnergyMonitor is configured as a de-coupled separate component to the rest of the *CloudMonitor* tool as each *EnergyMonitor* is specific to the PDU with which

it communicates. Some PDUs can be grouped together, such as those that require communication over SNMP, whereas others may require a custom protocol.

The PDU used by the St Andrews private cloud in this work communicates with the *EnergyMonitor* module using the Simple Network Management Protocol (SNMP). The PDU in this work acts as an agent within the system and responds to queries for data in real time.

We use a Rairitain PX-5367 PDU to power the servers in our experiments. It is possible to request energy usage information for those servers that are connected to the intelligent PDU by directly querying the factory provided SNMP API.

```
private int snmpCommand(String info) {  
    // Set up get active power snmp command  
    String cmd = "snmpwalk -v1 -c public -m ./MIB.txt "  
        + this.PDU_IPAddress + " " + info + "." + this.socket;  
    // Execute snmp command  
    return executeCommand(cmd);  
}
```

Snippet 3.2: *EnergyMonitor* preparing SNMP walk

Snippet 3.2, above, shows the Java method to prepare an SNMP walk command. The information requested is inserted into the command line parameters before the command is executed.

The *EnergyMonitor* module executes a series of *snmpwalk* commands requesting each of the dynamic power variables as defined in Table 3.4 in below. *snmpwalk* is a standard Unix command for interfacing with an SNMP device that collects a sub tree of SNMP management values from the target. Once the *EnergyMonitor* module has executed these commands the results parsed for insertion into the *CloudMonitor* data store.

An example of the power data collected at each 3-second interval would be:

Date	2013-08-28 22:43:03
Machine ID	DE:3E:D5:9B:22:CA
PDU Socket Number	7
Active Power (W)	85
Voltage	233
Current (A)	0.422
Watt Hours (W/h)	17372

Table 3.2: Example of *EnergyMonitor* data

If the power estimation functionality is to be used, the *EnergyMonitor* will, at each sampling interval, gather the most recent relevant hardware metrics and apply the store power model to them as a mathematical calculation.

To generate and store a power model two timestamps should be selected that correspond to the beginning and end of the *training phase*. *EnergyMonitor* will then generate the model through regression over the hardware values and corresponding PDU readings.

3.2.4 *MemoryMonitor*

The *SIGAR* library sometimes proved to be intermittently unreliable at accurately monitoring memory usage under testing. To combat this unreliability, a separate subcomponent was developed that executes the standard *Linux* command *free* to get accurate information about how much memory was being used by the system under observation.

```

// Run free command and get printout

ByteArrayOutputStream out = new ByteArrayOutputStream();

Process process = Runtime.getRuntime().exec("free");

DataInputStream input = new DataInputStream(process.getInputStream());

// Extract answer

String[] array = input.readLine().split(" ");

String used = array[array.length - 4];

insertIntoDB(used);

```

Snippet 3.3: Memory monitor gathering used memory data from Linux OS

Code snippet 3.3 above details the process by how *free* is invoked by the java program and how the answer is parsed from the system response. An example of the metrics collected are detailed below:

Date	2013-08-28 22:43:03
Machine ID	DE:3E:D5:9B:22:CA
Memory used (MB)	10501

Table 3.3: Example of *MemoryMonitor* data

This module is optional and not part of the required *CloudMonitor* software. Our tests found that on some Linux distributions the memory utilization values were more reliable with the *MemoryMonitor* module on.

3.2.5 Metrics collected

The static and dynamic data that is collected by *CloudMonitor* by the three modules that have been discussed are outlined in Table 3.3 and Table 3.4 respectively.

The schema of the *CloudMonitor* data store is outlined in the Figure 3.3 below:

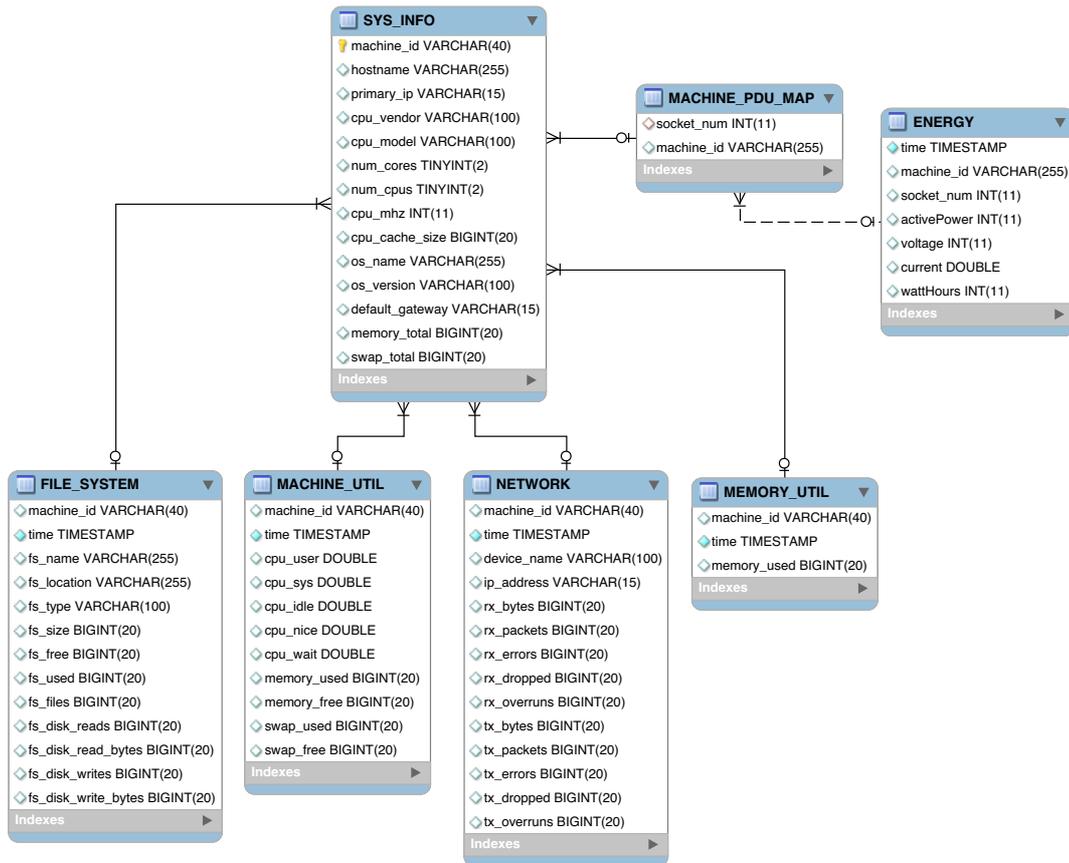


Figure 3.3: CloudMonitor EER diagram

The schema can be explained by breaking the data down into two categories: information that is static and does not change for each reading, for example the CPU speed, and information that does change, for example the CPU utilization, for which there is an updated value each polling interval.

There are 7 objects, or tables for which data is relevant to this thesis:

- *sys_info*: populated with static information about the host such as CPU vendor, clock speed and host MAC address.
- *machine_util*: contains data on the utilization of the physical host, including CPU usage percentages.
- *file_system*: contains utilization data over time for the hard drive usage of each host

- *network*: contains data on the network usage of each host, include bytes sent and received.
- *memory_util*: data on the memory utilization of each host over time
- *machine_pdu_map*: the mapping table to relate each power socket to a physical host
- *energy*: energy data is collected from a PDU or estimated by *CloudMonitor*, including the current active power drawn.

Table 3.4 shows the type of static information that is collected and stored in the *sys_info* table.

Monitored Target	Unit
CPU Vendor	Vendor Name
CPU Speed	MHz
CPU Cores	Number of cores
Hostname	Name
Operating System	Name & version number
Machine Identifier	MAC Address

Table 3.4: Static information collected by *CloudMonitor*

Information in the other tables is dynamic, and is identified by a timestamp and the machine identifier for the host the data relates to. The machine identifier is used as a primary key for most tables in the *CloudMonitor* schema and both static and dynamic values are updated when the tool is run on the

machine identified.

Monitored Target	Data recorded	Unit
CPU	Utilization average since the last 3-second time period.	%
Memory	Snapshot of amount in use at sample interval.	Bytes
Disk	Snapshot of amount in use at sample interval.	Bytes
	I/O operations since last sample interval	Number of ops
Network	IP Address	IP identifier
	I/O operations since last sample interval	Number of ops
Power	Active Power at current period.	Watts
	Voltage	Volts
	Current	Amps
	Watt Hours total since the power meter began recording	Watt Hours

Table 3.5: Dynamic information collected by *CloudMonitor*

For dynamic content, the machine identifier is used in conjunction with a timestamp to provide the key for that table. Values that have been identified for the current time period by the tool are inserted into the relevant table on each update.

Examples of the dynamic data are the resource usage metrics collected on the system subcomponents, the CPU, Memory, Hard Disk and Network attributes of the system. This data can be used to provide insights as to what each

machine is doing and the current utilization rate. These insights can help system administrators make decisions such as where workloads should be placed in a distributed computing environment.

3.2.6 Data storage

CloudMonitor runs as a daemon on every machine in the system that the user wishes to monitor, collecting data at periodical intervals and transferring it to the configured data store. *CloudMonitor* agents push data to the store once every three seconds by default, but this polling interval can be adjusted in the tool configuration. The default is set at three seconds because during testing this period was found to give good data granularity without significant overhead. Other intervals that we examined from 1 second, which was deemed to put too much strain on the data storage part of the system, to 5 seconds for which the data begins to lose granularity, were disregarded.

The data stored by *CloudMonitor* is made up of hardware resource usage and may also include corresponding PDU data for this machine. Collecting data from a PDU is an optional task and not required for the general running of the system, meaning each agent needs to be configured to do so. The applications are configured on deployment and if the relevant option is selected then the system will request data from the PDU and invoke the *EnergyMonitor* module.

The tool can be configured in two data storage architectures: centralized and distributed. In the centralized model a single data store is used to provide a unified view of the system data, which is useful in an environment where there is a small number of agents or where we seek the observer effect to be absolutely minimized. The data store in such a case might be placed on dedicated host or offsite and typically takes the form of a MySQL database. MySQL was used in the example in this chapter as the number of servers in this experiment was relatively low. In, larger-scale deployments and proper analysis of the scalability of each data store implementation would be required to determine the suitable application for the scenario. Figure 3.4 below shows the centralised architecture.

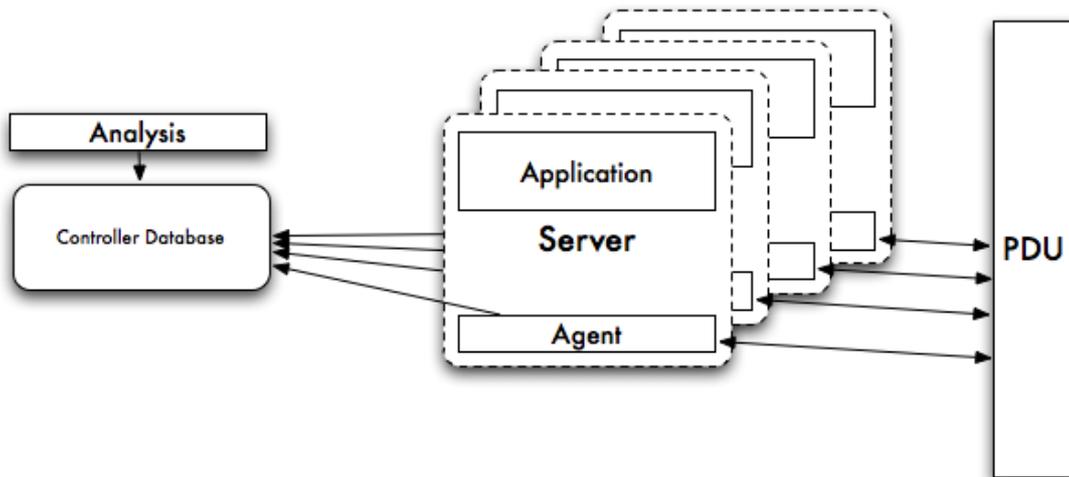


Figure 3.4: Centralized data storage architecture

This approach incurs the cost of transferring all collected information to the data store as soon as it is generated. With the default-polling interval for *CloudMonitor* set to 3 seconds this means that every three seconds, data from every agent in the system is being transferred to the centralized store leading to the possibility that it might be overwhelmed. However, the advantage of this eager sharing of information means that lookup times and analysis can be much quicker as the data is in one central location. This would be useful in situations where real time decisions are being made based on the monitoring stream.

The second approach is distributed, where alongside *CloudMonitor* each machine also runs a local Apache Derby SQL database instance to maintain state stored locally on the machine that is being monitored. Analysis can then be done by collecting information from the machines directly, resulting in a slower lookup than the previous architecture but without the overhead of additional, perhaps unnecessary network traffic as all information about each machine is kept locally.

Figure 3.5 below shows this architecture.

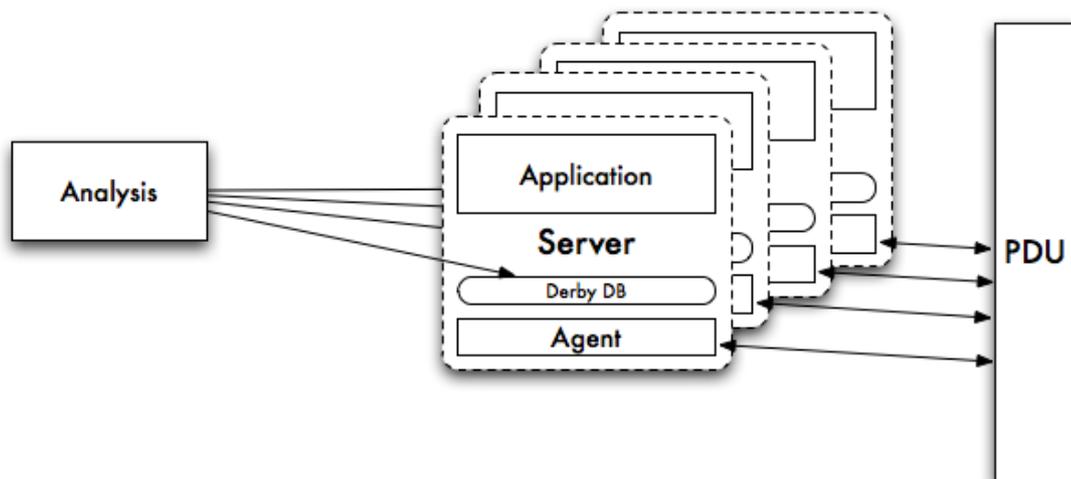


Figure 3.5: Decentralized data storage architecture

The switch from centralized to decentralized architecture can be set by the system administrator in the *CloudMonitor* configuration depending on their deployment preferences.

Preliminary analysis of *CloudMonitor* performance suggests that data sent and received from the data store averages about 1.5kB/s per daemon. The decentralized architecture would be appropriate once the number of machines being monitor is sufficiently large that network performance on the local network would begin to be impacted by the network traffic to the centralized data store. A suitable crossover point could be once monitoring traffic exceeds 1% of the available bandwidth, for example on a 1Gb/s local area network this would be once monitoring traffic exceeds 10Mb/s. A 1Gb/s LAN should therefore be able to support approximately 80,000 *CloudMonitor* daemons. This crossover point is a suggestion; the actual value should be determined for each deployment depending on the local requirements.

3.2.7 Portability and footprint

The agent programs are written in the Java programming language, to provide portability, allowing deployment on any platform with a Java VM implementation. While Java may not be the language with the lightest footprint, its ease of deployment far out-weighs any performance penalty.

Coupling the *SIGAR* library with the Java programming language allows

CloudMonitor to be deployable on Linux, FreeBSD, Windows and Mac OSX across a variety of versions and architectures without changes to the application code.

3.3 *CloudMonitor* evaluation

CloudMonitor was evaluated by instrumenting a typical web application as it runs through a 24 period of workload. The monitoring tool was deployed to collect information on the resource usage and energy usage statistics. These metrics are then used demonstrate the effectiveness of the energy estimation capabilities of the tool and the potential for an energy tariff for a typical bill-per-use cloud platform.

3.3.1 Experiment Description

We developed a video sharing web application to evaluate *CloudMonitor*, and developed a client-side tool that mimics users by performing typical user behaviour for a video sharing website - uploading videos, viewing the list of recently videos and watching available clips. The example web application and user behaviour tool were developed jointly with another PhD student and used in multiple experiments including this one.

The experiment uses a hypothetical web application that is deployed and monitored while usage patterns are applied to investigate its resource usage. The application was selected as it represents a typical web application that requires front-end nodes that run web servers, a storage node, and several worker nodes that perform any back-end processing. The video-processing application and the client-side tool are open-source and can be downloaded from [GitHub.com](https://github.com/alikhajeh1)²¹.

A synthetic video processing application is used here instead of specific benchmarks to be representative of a real-world cloud application. Each

²¹ <https://github.com/alikhajeh1>

component of the application displays the characteristics of a typical real-world scenario: web servers, storage servers and processing servers. The video processing web application allows users to upload video clips to a website. The application converts the uploaded clips to MP4 and OGG formats using the FFmpeg video conversion tool, in addition to making a JPEG and PNG snapshot for use as thumbnails.

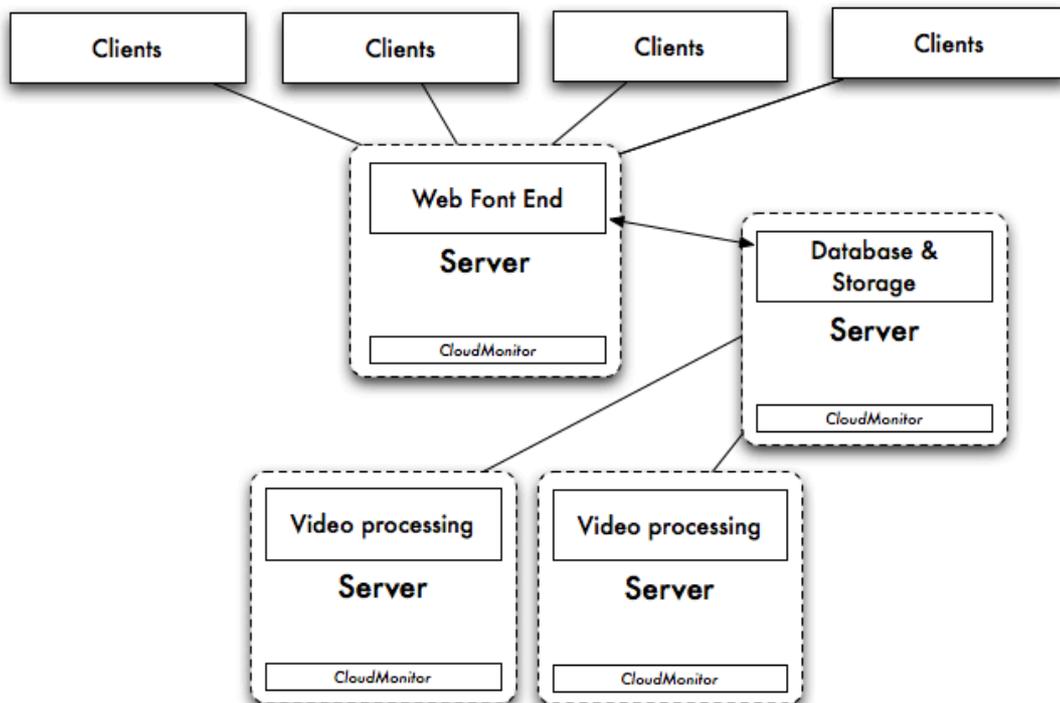


Figure 3.6: Video Processing application architecture

Our experimental setup for the application used four modern physical servers. The web server was deployed on one server, one server was setup as the database/file storage node; the remaining two servers were dedicated to video processing. Worker server 1 attempted to complete all background tasks, with worker 2 picking up slack when required. Figure 3.6 shows the system architecture of the video processing web application that we developed.

The Ruby on Rails framework was used to develop the web server and MySQL was used on the storage server. The web server moves the uploaded video clips onto the storage server and creates an entry in a queue with the job details of the video that is to be processed. The worker servers poll the queue once every second and run the actual video conversion job; converted files are

then moved back onto the storage server.

In order to provide a realistic workload to the web application, a client-side tool was created to simulate the browsing behaviour of many people using the web application. The client was written in the Python programming language to perform typical user behaviour when browsing a file sharing website; uploading a video, viewing the list of recently uploaded videos and watching available video clips. These actions are simulated in such a way as to produce the desired execution on the processing server, not the client.

For example, the task “watching a video” is completed by requesting a video clip from the server, which causes a network transfer. Actually playing back the file for a fictional user is not required as the server’s actions on each file request are completed once the file has been completely transferred. There was no requirement for more complex modelling of user behaviour as the intention of this experiment is to stress the hosting server as strongly as possible, which would not increase by introducing more varied browsing habits.

Two instances of the client program are used in this experiment, each hosted on individual virtual machines on the StACC private cloud. Each VM was of type m1.small –meaning they had a 1 virtual CPU core (host machine has Intel Xeon processors) and 512MB of RAM. Figure 3.6 details the layout and interaction between clients and the web application deployment.

3.3.2 Experimental results

The experiment that we ran simulated a single day of usage of the typical web application. A 24 hour period was chosen to provide the experiment with sufficient data to analyse the web application operations. All graphs have sample index as time on the x-axis. There are 86,400 seconds in 24 hours. We sample at 3-second intervals giving 28,800 samples for the period.

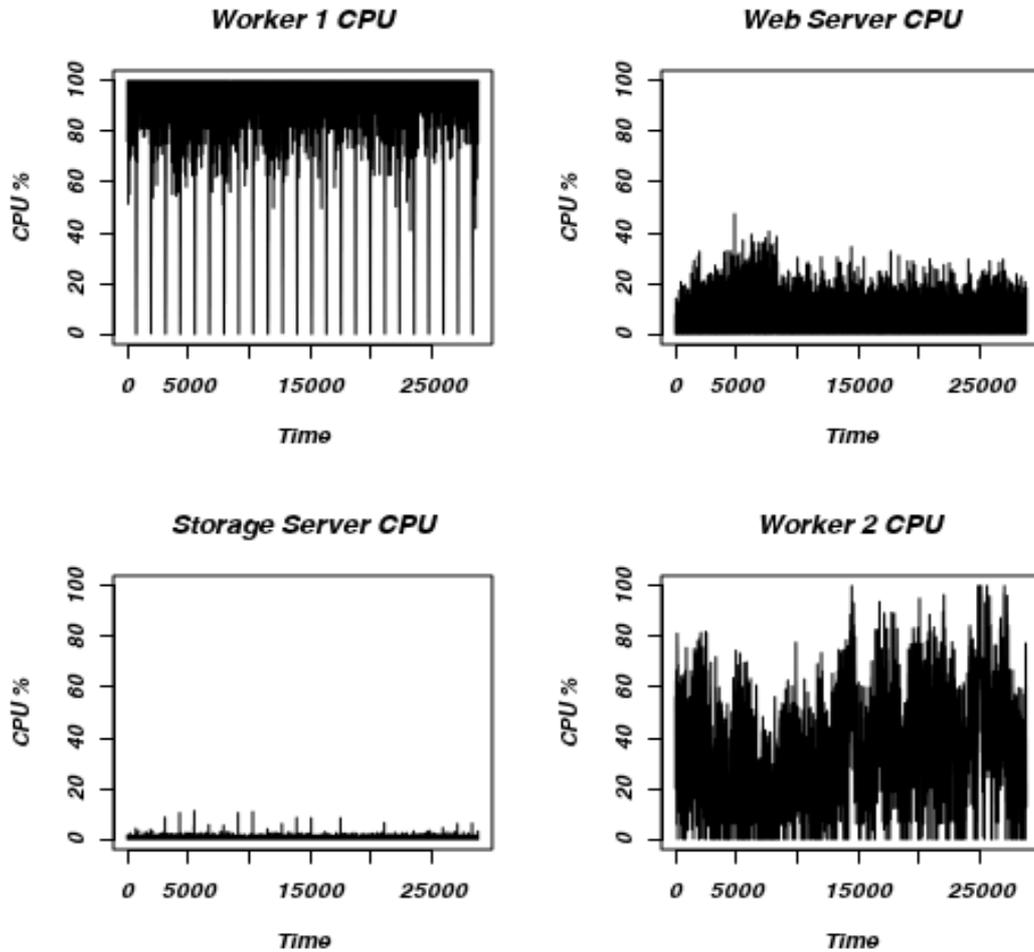


Figure 3.7: CPU usage for the 4 servers. Vertical axis is CPU % and Horizontal is time in 3 second intervals (maximum ~29k). Graphs are black data on white background.

As expected, Figure 3.7 shows that the first worker server displays a consistently high CPU usage, suggesting that it was almost fully utilized throughout the experiment. Worker 2 used a highly variable amount of CPU suggesting that its workload was irregular as it met demand for the video processing. The web server used a low, but irregular amount of CPU and the storage used very little CPU over the monitored period. Worker 1 CPU had downward spikes to 0% CPU a number of times over the monitored period. This would have occurred when in the short period when a download was complete and a new one had not yet begun.

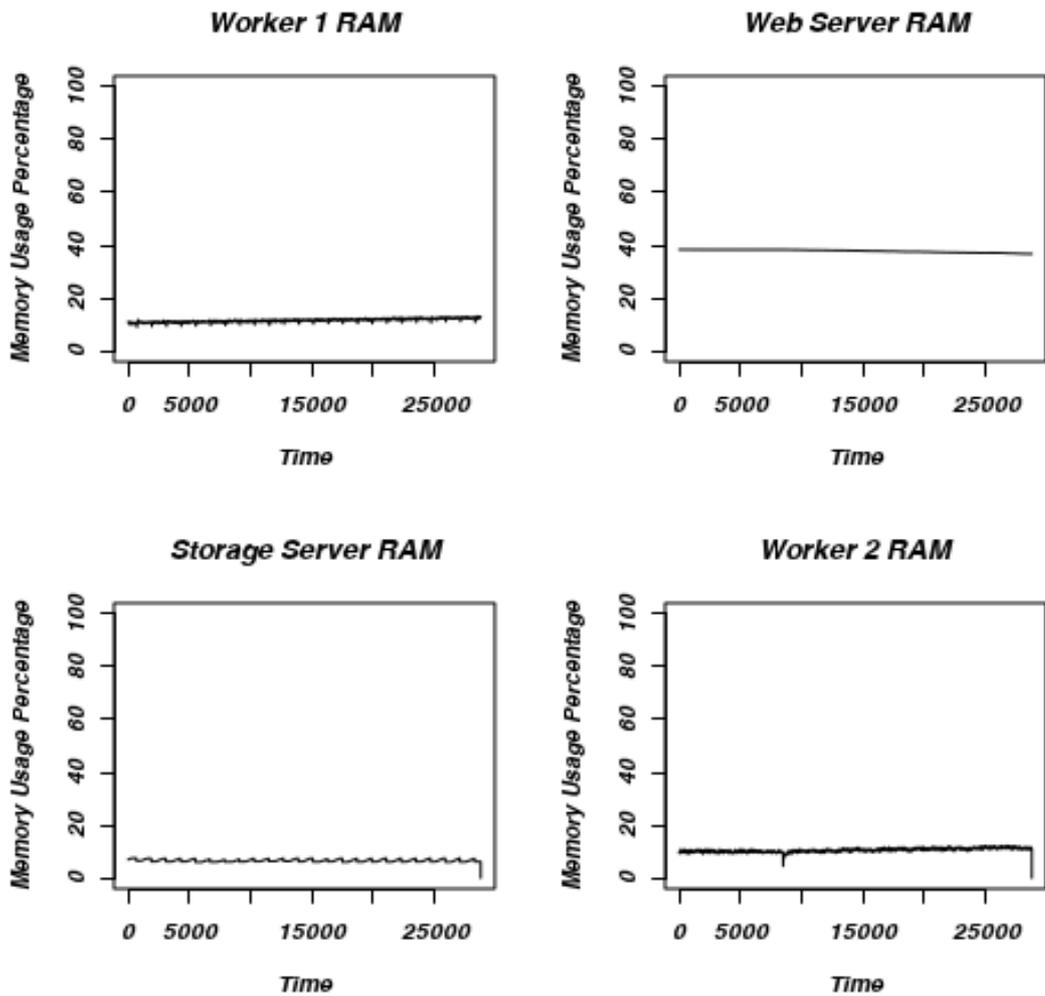


Figure 3.8: RAM usage in percentage over 4 servers. Vertical axis is Memory usage % and Horizontal is time in 3 second intervals.

Figure 3.8 displays the memory usage for the servers of the same period. The workers use a small, but consistent amount of memory. The web server uses the most RAM, which is consistent with its task of serving dynamically generated web pages. The storage server's memory usage fluctuates around a low level.

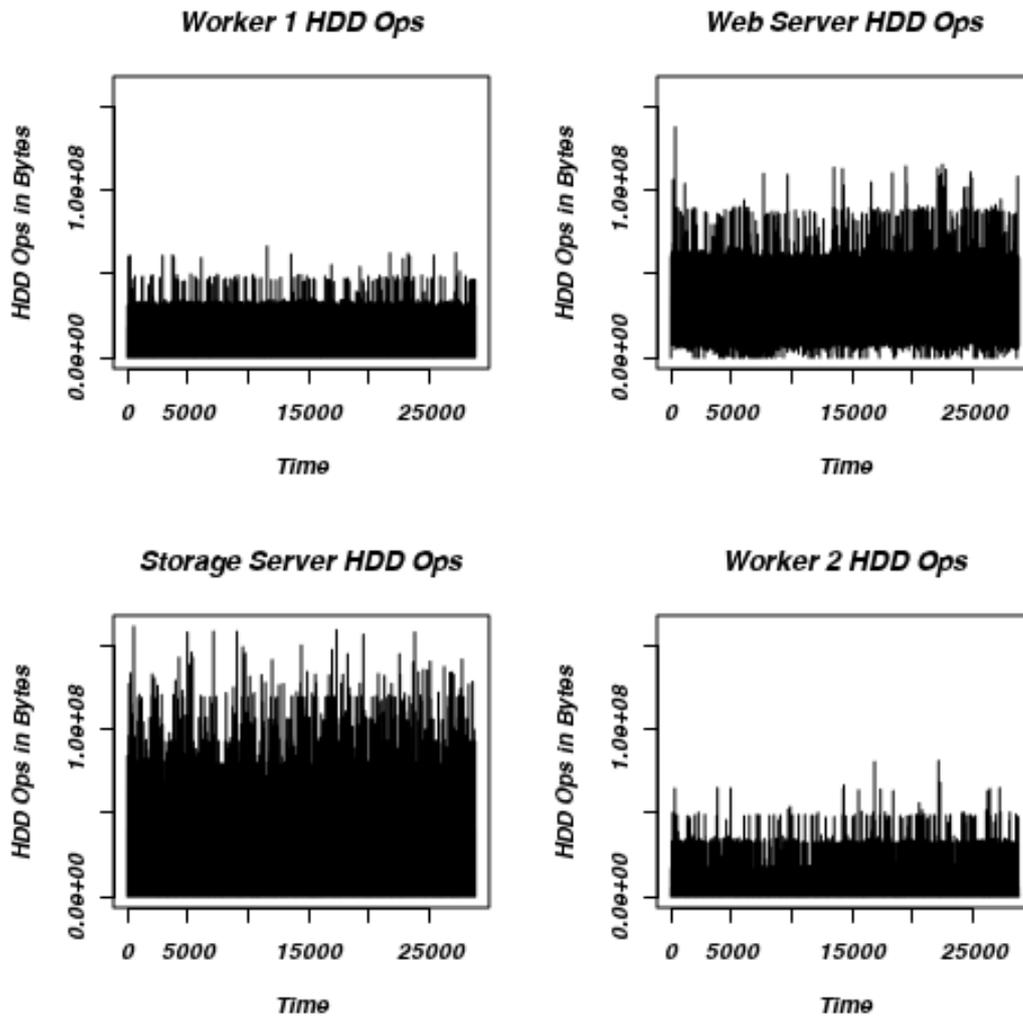


Figure 3.9: DISK usage in Bytes for the four servers. Vertical axis is Hard drive I/O ops in bytes transferred per 3 second interval and Horizontal is time in 3 second intervals.

In Figure 3.9 we begin to see insights that can only be discovered from a multi-dimensional study of resource usage that *CloudMonitor* provides. The storage server, which to this point has appeared to have very little resource usage, is in-fact conducting thousands of bytes worth of read and write operations per interval sample. This graph shows disk I/O operations in bytes/three-second intervals. It is clear that the storage server is in fact I/O bound. However, by examining Figure 3.10, a different picture begins to emerge:

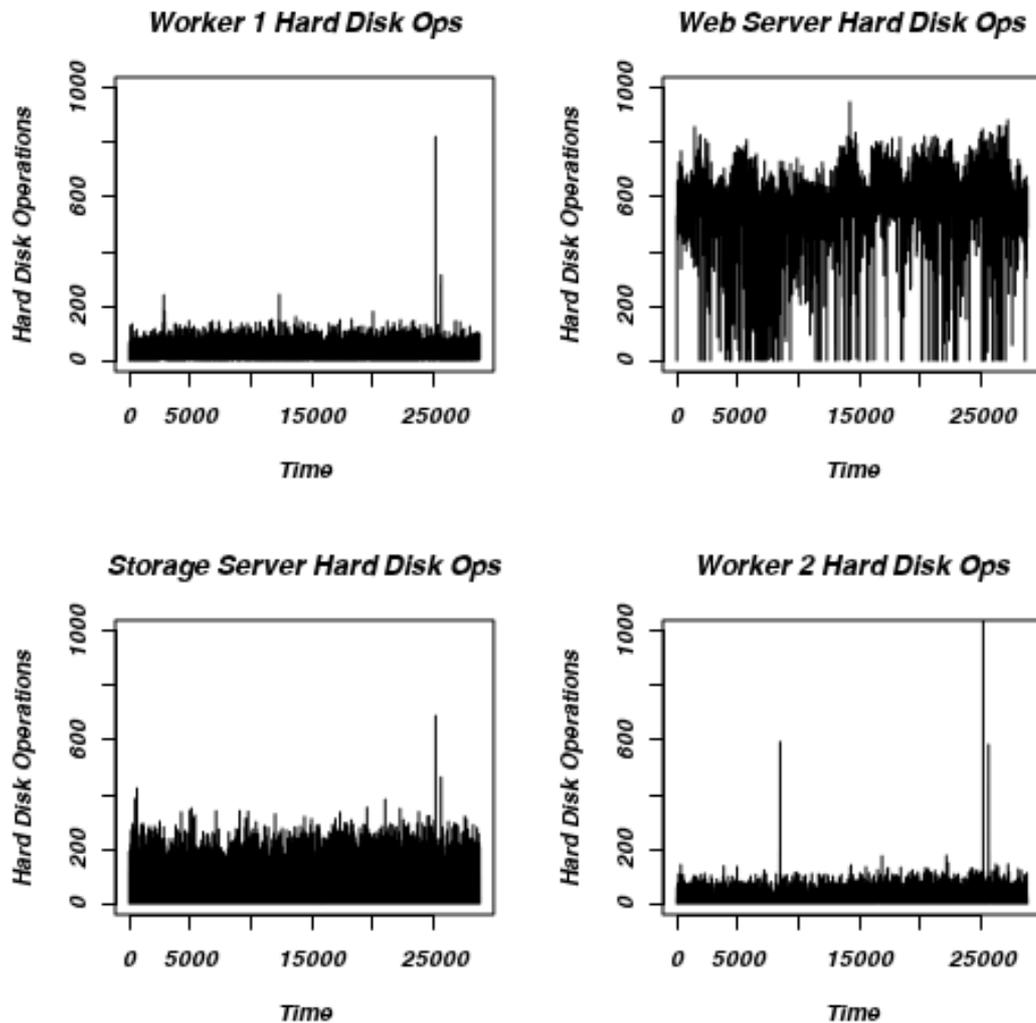


Figure 3.10: Disk usage in operations per second. Vertical axis is Hard disk operations per second and Horizontal is time in 3 second intervals.

Figure 3.10 shows the read and write operations per interval for the same servers, over the same period. Now it is clear that while the storage server is reading and writing more data per operation, the web server node is actually conducting the most operations. Over the 24 hour period the web server conducted over 16 million I/O operations, compared to the storage server's 1.7 million. The number of I/O operations per server is very important when evaluating the future deployments of such a system, as cloud service providers may charge explicitly per operation.

Therefore, when considering deployments, the number of I/O operations can have a significant impact on the financial cost of deployment. From analysis of this data, we might conclude that the performance of the web

server would benefit from a solid-state hard drive with faster random access times, this is not central to power measurements but is an insight that is gained as a by-product from the increased visibility that *CloudMonitor* gives administrators into their running applications.

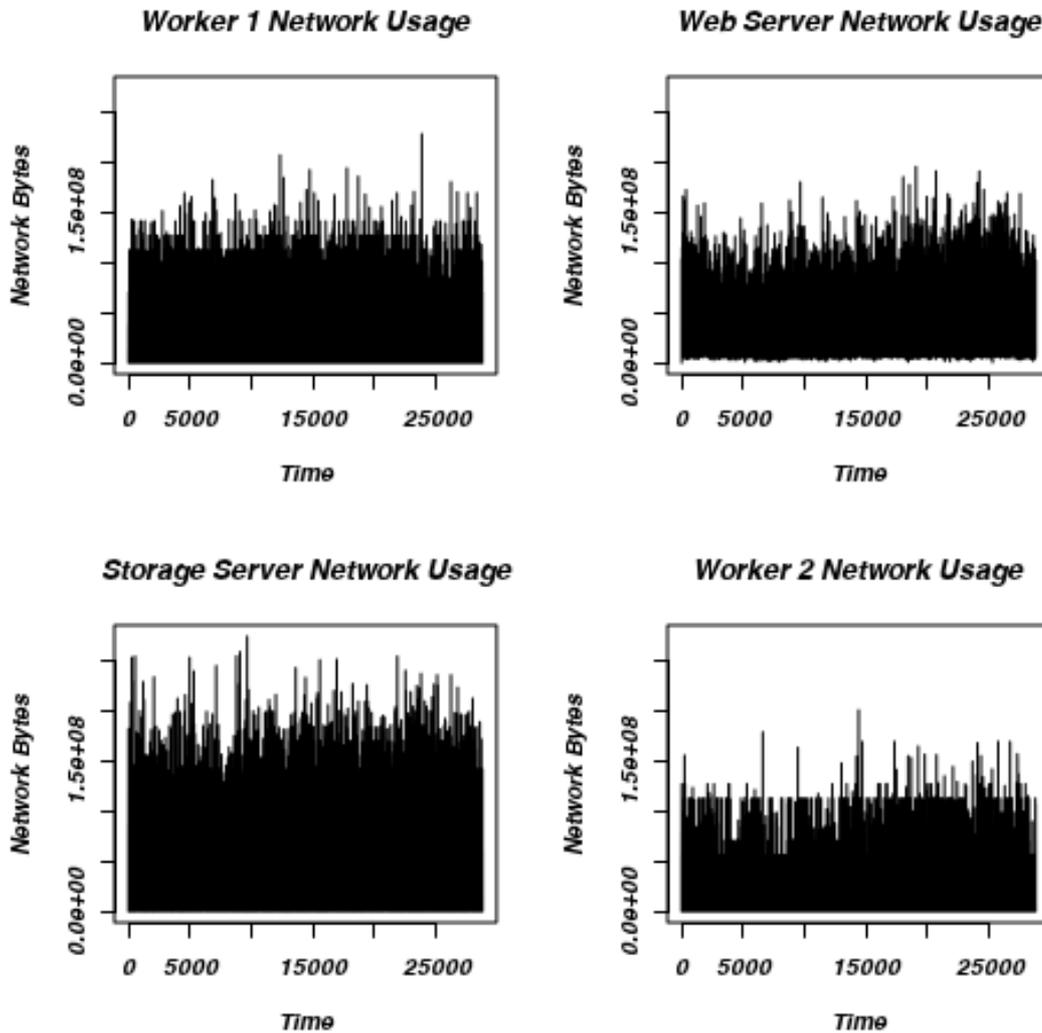


Figure 3.11: Network usage in Bytes over the four servers Vertical axis is Network usage in bytes per sample interval and Horizontal is time in 3 second intervals.

Figure 3.11 displays the network usage in bytes per sample interval for the application components. As expected, the storage server appears to be using a large amount of network traffic (which is entirely internal to the local area network between the nodes) and we can also visualize the external traffic through the web server. In total, the webserver transmitted 481GB over the 24-hour experiment and received 251GB in requests and file uploads.

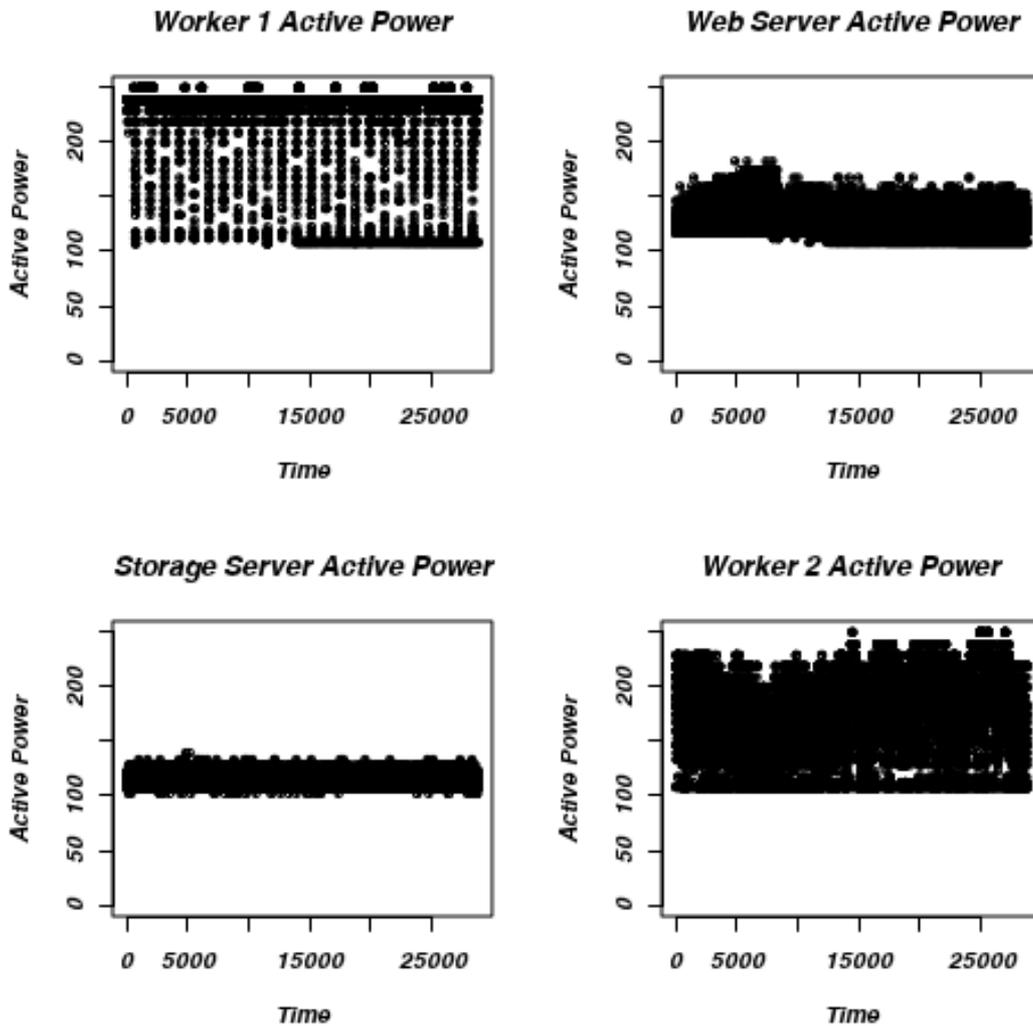


Figure 3.12: energy usage in Watts over the four servers. Vertical axis is Active power drawn in Watts at each sample and Horizontal is time in 3 second intervals.

Each server in our experiments was connected to a socket on a PDU. Figure 3.12 shows the power levels used by the servers are dominated by their CPU usage. However, that cannot explain all the fluctuations, as for example the storage server used an almost constant, low amount of CPU yet its energy usage fluctuated across a band between 100 and 150W. The data provided by *CloudMonitor* would suggest that the HDD usage characteristics of the application could be responsible for the power fluctuations.

As discussed in section 3.2, the *CloudMonitor* power model is as follows:

$$Power = \alpha + (\beta_1 * CPU) + (\beta_2 * Memory) + (\beta_3 * Hard Disk) + (\beta_4 * Network)$$

The next stage of the experiment was to define the 24 hours worth of resource

usage data collected by *CloudMonitor* as the *training phase* and perform linear regression analysis to obtain the coefficient weights that should be used to tailor this power model to the hardware configuration.

The time-series records accumulated by *CloudMonitor* during the experiment were evaluated using the power model generation functionality and validated using the *R* statistical package. Samples at each timestamp were taken to be independent without a delay model.

Component	Model symbol	Coefficient value
Baseline Power	α	107.5
CPU (percentage between 0 and 1)	β_1	124.9
Memory (in MBs used)	β_2	5.471x10-06
Hard Disk (in operations per sample interval)	β_3	3.661x10-02
Network (in bytes received and transmitted per sample interval)	β_4	3.382x10-08

Table 3.6: Linear regression analysis

First, analysis is required to see if there is indeed any correlation between the resource usage (CPU, memory, hard-disk and network) and energy consumed. Applying a Welch Two Sample t-test to the data results in a p-value smaller than 2.2×10^{-16} – indicated that is very unlikely that these results could have arisen from sampling variability. Therefore, we adopt the explanation that the energy consumed by these devices is indeed somehow related to the combination of CPU + memory + hard-disk + network.

Performing linear regression analysis results in the formula outlined below. Each of the independent variables in the equation had p-value of less than

2×10^{-16} , suggesting high significance in the equation.

The power estimation model, described in Section 3.2, becomes for this configuration of servers:

$$\begin{aligned} \text{Power} = & 107.5 + (124.9 * \text{CPU}) + (5.471 \times 10^{-06} * \text{Memory}) \\ & - (3.661 \times 10^{-02} * \text{Hard Disk}) + (3.382 \times 10^{-08} * \text{Network}) \end{aligned}$$

It is worth noting the order of statistical significance of each of the variables. CPU has the most impact on power usage, followed by Hard Disk, Memory usage and Network. This is consistent with our intuitive understanding of how modern computers operate, with spinning disks of the hard drives having the second biggest effect on energy usage behind CPU. It therefore suggests that a move towards solid state hard drives could have a significant impact on the energy consumption of server hardware.

Memory and Network coefficient values are small, but they are based on megabytes and bytes for their respective components. Therefore it is likely that the bytes value would be high and the coefficient would be smaller to compensate. In contrast, the CPU value would be between 0 and 1 to indicate percentage and as a result its coefficient is a large positive number. Each of the coefficients, regardless of size, are valuable in the model's ability to comprehensively measure the energy impact of different workloads. The power usage of a workload with high utilization of a subcomponent that has a small coefficient can still be accurately estimated.

The calculated intercept of 107.5 of the linear regression correlates strongly with the observed idle energy consumption of these particular servers at around 108W, confirming that it is the value of the *baseline* power. The *dynamic* energy consumption above this value is dependent upon the amount of work applied. In this case, the *baseline* power value is 43% of the observed *peak power* of 249W.

To evaluate this power model, we began a second, similar, experiment to that described above. Workload rates on each of the 4 servers were varied to show that the power model is applicable under different workloads. While we used the same hardware (as required by the premise of a power model relating to a

particular hardware configuration) the workload on each individual machine was varied to test the robustness of the model.

Applying the generated power model, above, to the resource usage for the evaluation experiment resulted in an average error rate of just 3.91% when comparing the predicted Active Power values with the actual PDU values. This low error rate shows that a generated power model can compute active power values that are close to the measured energy consumption.

3.3.3 An energy tariff for cloud computing

Accurate energy measurements help to improve the performance of systems by providing insight into the energy consumption of different aspects of the system but it can also lead to the possibility of unbundled energy pricing. If an organization bills its users or departments according to their resource usage of a private cloud then it will be possible using a tool such as *CloudMonitor* to accurately provide an estimation for energy usage, allowing it to be billed separately. Explicitly exposing the cost of energy to software developers might encourage them to reduce the energy footprint of their applications.

The video processing web application needs 15.73kWh²² every day to operate for the workload levels applied. The University of St Andrews is charged a value of £0.089/kWh by their energy provider. At the current rate, including a 15% annual increase as discussed in Chapter 1, the overall cost for electrical power over a 36-month period would be £1,743 for the video processing application. Of course, this is assuming that load of the application is static for 24 hours each day. In a real world scenario the workload would very likely vary over a day or yearly cycle resulting in a variable power load and then a different cost for electricity.

Khajeh-Hosseini *et al* [91] have developed tools to support decision makers during the adoption of cloud computing in their organizations. One tool is a cost-modelling application that is available from PlanForCloud.com. The data gathered during the experiment can be input into this tool to get a cost

²² Total of kilowatt hours used for each server in the experiment over the 24 hour period.

estimate of deploying the application on a public cloud, in addition to a breakdown of the cost categories.

Assuming our university was to set up a private cloud and use a similar pricing scheme as the Amazon Web Services EU cloud but with an additional charge for energy usage, Table 3.7 shows how much the hypothetical private cloud would charge for the video processing application over a 3-year period. As expected, data transfer and VM instance hours account for the majority of the costs. However, it is interesting to note that energy usage would account for 2.6% of the total costs, a higher value than storage and storage I/O request costs. The Amazon cloud costs probably already include an implicit charge for energy. If this was unbundled, the percentage of costs for energy usage would be even higher.

Category	Cost
Data Transfer	£36,581
Virtual Machine Hours	£25,550
<i>Energy Usage</i>	<i>£1,743</i>
Storage	£1,464
Storage I/O Requests	£1,444
Total	£67,052

Table 3.7: Cost forecasts for test system

These values shows that, by using a tool such as *CloudMonitor*, cloud providers can, if they choose to, could charge users for the energy consumption of their deployments. In turn, software developers can estimate those costs effectively using only their application resource usage and the power model for the system they will be deployed upon.

The costs in Table 3.7 assume that 4 Heavy-Utilization 3-Year Reserved Standard Extra Large instances, each with 146GB of EBS storage are used (this is a similar specification to the physical servers that we used). *CloudMonitor* showed that our video processing application would have 14,430 GB/month data downloaded from its web server, and 7,530 GB/month data uploaded to its web server. In addition, the number of disk I/O operations/month for the servers was as follows: web server: 492 million, processing server 1: 21.4 million, processing server 2: 10.7 million, DB/storage server: 51.9 million. It would have been difficult to obtain this detailed resource usage data without using a tool such as *CloudMonitor*.

3.4 Conclusion

CloudMonitor is a distributed monitoring tool suitable for gathering information about private cloud deployments. It can provide precise information about resource usage and can estimate energy usage once a power model has been trained on a particular hardware configuration. We have improved on the related work by automatically creating an accurate power model using a live linear regression approach on data from an accurate PDU. This automated approach shows a mean accuracy value of 96.09% when evaluating different workloads on the same hardware.

The tool is scalable as it is implemented entirely in software with only an initial training period requiring hardware PDU. Once a power model has been trained on a particular hardware configuration it can be rolled out across all machines in a datacentre of the same configuration. It is lightweight, with CPU usage of less than 1% and network traffic of 1.5kB/s per daemon. There are two designed architectures; centralized and decentralized, the first of which can support up to 80,000 machines before switching models for further scalability.

By instrumenting their servers with *CloudMonitor*, administrators gain increased insight into their systems resource usage and energy consumption

leading to the possible identification of issues or general areas that can be improved or made more efficient. On the set of hardware used for the evaluation experiment, inputting the average CloudMonitor hardware utilization values into our power model gives an estimate power cost of 1.25W for the tool's usage on each host. This value is 1.1% of baseline power value of these hosts and well within the mean margin of error for CloudMonitor. Therefore the observer effect of the tool is minimal.

This work has also shown that the development of an energy tariff for utility computing systems is possible, by accurately estimating energy usage based on the amount of resources used by each user or department. Such a tariff might incentivize software developers to create energy efficient applications, further increasing the efficiency of the system.

The *CloudMonitor* software can be downloaded from <http://jws7.net/CloudMonitor>.

4 Workload mixing

4.1 Introduction

Demand for computation is increasing faster than we can reduce the amount of energy required to do each task. In spite of more efficient, energy-proportional hardware, the increasing demand means that overall energy usage is likely to continue to increase in future unless active steps are taken to reduce the energy used by our computing systems.

Organizations are motivated to minimize their energy usage to reduce costs. For example, the Computer Science department of the University of St Andrews, which is a small organization, spent over 20,000 kW/h in both June and July 2013 at a cost of more than £2,500 per month solely on computing infrastructure. This extrapolates to an energy cost of £30,000 per year.

One approach to reduce energy consumption is to utilize software techniques to reduce the amount of energy consumed by hardware. However, this challenge is not simple. Traditional approaches to software based energy conservation attempt to minimize the number of computing devices used and power down those that are idle. Such approaches may impact dependability if VMs are consolidated onto a low number of VMs, as the loss of a physical host will result in the failure of a larger percentage of VMs.

The literature review of this area in Chapter 2 shows that in the simplest terms, there are two approaches to traditional scheduling algorithms for a cloud; ‘spread’ and ‘stack’. Spread balances workload across the available hosts, ensuring no host is overworked and therefore achieving the best possible application performance. Round-robin assignment is an example. *stack* is the opposite of this strategy, where jobs are stacked on the least number of hosts with the aim of reducing the amount of resources required to achieve the body of work and powering down those compute nodes that are not required. Bin packing is the traditional approach here.

A bin-packing strategy where work is condensed onto as few hosts as possible

is generally presented as an energy efficient solution to workload allocation. As we have seen that even lightly loaded physical hosts consume significant amounts of energy, lowering the number of physical hosts employed in an allocation strategy is an efficient approach for reducing energy consumption.

However there a number of issues that this strategy presents that may not be acceptable to some organizations:

- Risks to dependability: Bin packing is risky when dependability is a key desired system property. Often administrators may deploy redundant copies of software applications and balance workload requests to them. In that case, certain tasks must be spread throughout a datacentre to ensure that a single hardware failure, such as a corrupt hard drive or faulty router, do not compromise a large percentage of the application redundancy.
- Over-working of hardware: over working or repeatedly power cycling hardware through increased consolidation, constant workload migration, or power cycling may damage and reduce the lifespan of hardware components[35][75].
- Reaction time: when the workload demands are varied computational hardware must be powered on and made ready to react to the incoming workload. This may be automated in a complex system deployment but may also involve having staff on standby to react to demand.

Therefore, reducing power demand in a datacentre by reducing the number of physical servers powered on at once is unlikely to be an acceptable option for many data centre operators. Servers, realistically, are usually powered on and operating for essentially all of their recorded life [75]. As servers typically consume up to 60% [41] of their peak power when idle a significant amount of energy is still being consumed. This energy usage is considered to the datacentre "base load".

With this assumption, the aim of our work, therefore, is to minimize dynamic energy consumption through software so that it is as close to the base load as possible and rely on other technological changes, such as more power efficient

processors, that may come in time for base load reduction.

To achieve our goal of energy reduction without server power cycling, a different approach to workload allocation must be adopted that can take into consideration the need for workload spreading to maintain dependability. This new approach must attempt to reduce energy consumption as much as possible while still maintaining system requirements and restrictions. To do this we evaluate the allocation of workloads in different configurations to understand the trade-offs between power and performance.

Workload Mixes are an approach to application consolidation that is respectful of both performance and energy consumption. This differs from bin packing, which solely focused on application consolidation to reduce energy usage by maximally utilizing n-dimensions of the available computing hardware. Such an approach has no concept of the ramifications that might occur when collocating jobs of particular characteristics.

Similarly, workload mixes differ from traditional load balancing which takes a “dumb” approach to satisfying performance targets. Load balancing is predicated by the assumption that lighter overall load on each host will result in better application performance, regardless of job type.

A Workload mix will attempt to look for complementary attributes amongst the profiles of the applications to be allocated. A profile is made up of the resource usage data for a running application. Complimentary profiles are profiles that, when paired, may result in increased performance or reduced energy consumption. The experiments outlined in this chapter show that such complementary attributes are possible, and that by exploiting these attributes it is possible to influence the characteristics of the overall system.

This evaluation has been conducted over a series of experiments that will now be outlined. The lightweight software based monitoring application, *CloudMonitor*, outlined in the previous chapter, was used to collect data for all the following experiments.

4.2 Experimentation

Experiments were conducted to investigate the effect of workload mixes on energy consumption in a computing environment. If workload mixing can be demonstrated to have an effect, then it may be possible to find an optimal workload mix, which can be consistently demonstrated to require less energy to complete the same amount of work.

To that end, we conducted a series of five experiments, beginning with a *null hypothesis* experiment to demonstrate that workload mixes do have an effect the energy consumption of a system.

The five experiments conducted in this chapter are outlined in the table below:

Experiment	Description
Null hypothesis	Disprove that workload mixes have no effect on energy.
Virtualization effect	Introduce a virtualization layer and confirm that energy is still affected by the workload mix
Performance vs. Energy	Examine the effect workload mixes have on application performance
VM sizes	Investigate the effect of different VM configurations on performance and energy consumption
Mixes vs. bin-packing	Compare the operation of a workload mix and a bin-packing strategy.

Table 4.1: List of experiments described in this chapter

The experiments were all conducted over the same fixed period of time to ensure that only the mix, and not job completion time affected energy consumption. Jobs were looped for a fixed period and the energy consumption of the hosts for that period was measured.

4.2.1 Null hypothesis experiment

The Null Hypothesis is that workload mixes have no effect on the energy consumption of a private cloud system.

4.2.1.1 Experiment design

The experiment involves four tasks of two types executed concurrently on two compute nodes for one hour.

In a virtualized environment such as private cloud system, tasks will typically outnumber computing hardware nodes. In this experiment there are two types of task, one that is CPU intensive and one that is disk (DISK) intensive. These represent real world work such as a mathematical experiment that is CPU bound and a heavy-use database application that is I/O bound.

The CPU intensive task is designed to generate computational threads that will employ each multi-core CPU on the host for the time required. The DISK intensive task writes a random integer to a random selection from 10 files, each of which exceeds the CPU and DISK cache size (12MB and 16MB respectively). These workloads are entirely synthetic and are intended to simply exercise the relevant resources for the time required. The code used for these workloads can be downloaded at github.com^{23 24}.

The underlying hosts are two hardware nodes, both 2010 model Dell *Poweredge R610* servers with 2 Intel Xeon E5620 Processors, 16GB of RAM and a Seagate *Savvio 10K 6-Gb/s 146GB* Hard Drive. The hosts run Ubuntu 12.04 OS.

Four tasks, two of each type, are executed under two workload mixes. For the co-located mix tasks of the same type are co-located on a single host – CPU intensive tasks are located on Host A and DISK intensive tasks are located on Host B.

²³ <https://github.com/jws7/experiments/tree/master/CPULoad>

²⁴ <https://github.com/jws7/experiments/tree/master/HDDActivity>

The split mix splits tasks apart and puts one CPU task and one DISK task on each host.

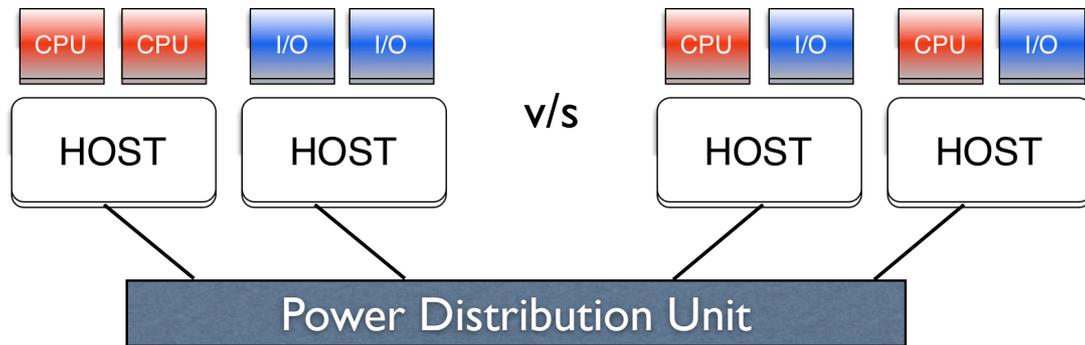


Figure 4.1: Diagram of first experiment in workload mixing

To limit the interference of mix runs, a square wave usage pattern was adopted. The hosts began in an idle state prior to each stage of the experiment and were allowed to return to this idle state for at least 15 minutes between runs. This allow the hosts to “cool down” and return to their idle energy consumption levels. Each experiment ran for one hour.

Data for the experiment was collected in real time using the *CloudMonitor* software application. Hardware utilization data was collected from each virtual machine (when used for later experiments) and host. In addition, for increased precision data from a billing-level power distribution unit (PDU) was collected to monitor the hardware energy usage. The data below is the mean watt-hours used by the experiments over three runs of each mix (12 compute hours in total). The energy data is collected as the difference between the cumulative watt-hour readings for the power adaptor connected to each host before and after each experiment run. The data is stored in a MySQL database unless otherwise noted and is processed using Microsoft Excel and R, the statistical package for the sciences.

4.2.1.2 Results

Mix	Host A	Host B	Total
Co-located	$196 \pm 0 \text{ W/h}^{25}$	$128 \pm 0 \text{ W/h}$	$325 \pm 0 \text{ W/h}$
Split	$172 \pm 0 \text{ W/h}$	$165 \pm 0 \text{ W/h}$	$337 \pm 0.82 \text{ W/h}$

Table 4.2: Mean energy usage results of Null Hypothesis experiment (3 runs)

The mean watt-hour values for this experiment are shown in the table above. The experiment was repeated three times and on each occasion the individually reported figures varied by no more than a single watt-hour from the mean.

Analysis of these results to determine their statistical significance revealed a p-value of 2.45×10^{-7} from a Student T-Test, so we conclude that it is very unlikely these results could have arisen from a sampling variability. We therefore adopt the conclusion that the workload mix affected the energy consumption and so reject the *null hypothesis*.

A Student t-test is appropriate to use here as we have a clear nominal variable, the mix used, and a measurement variable, in this case Watt-Hours, that we wish to measure. For all other similar measurements in this thesis, where two mix results are compared to ascertain a difference, the same test is used.

It is clear from the table above that the two mixes produced different energy usage totals. The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 96.4% of the energy required for the non-optimal mix (split). The *null hypothesis* has therefore been disproved.

4.2.2 Virtualization effect experiment

This second experiment introduces a virtualization layer to the same experimental design as before. The aim of this experiment is to confirm that

²⁵ Standard deviation variance used in all experiments.

the conclusion of the first experiment, namely workload mixes have an effect on energy consumption, is still valid when a virtualization layer is introduced.

4.2.2.1 Experimental design

Virtual Machines were created using QEMU on Ubuntu hosts. Each virtual machine was assigned approximately one half of the host's physical resources. As each virtual machine held only one task, the resources given to each VM were designed as far as possible to mimic the previous experiment. Therefore each virtual machine was assigned 4 CPU cores and 8GB of RAM.

A further experiment (Section 4.2.5) takes into consideration the effect of different virtual machine size on energy consumption.

As before, there are two types of task, one CPU and one DISK intensive. The software tasks remained the same, as did the underlying hardware and monitoring solutions.

The experiment involves four tasks (two of each type) executed concurrently in single virtual machines for one hour in two different mix configurations. As before, In the co-located mix, co-located virtual machines are given tasks of the same type – CPU intensive tasks are located in VMs on Host A and DISK intensive tasks are located in VMs on Host B.

In the split mix the similar tasks are split between both hosts, therefore, Host A and B will now have VMs running one CPU task and one DISK task.

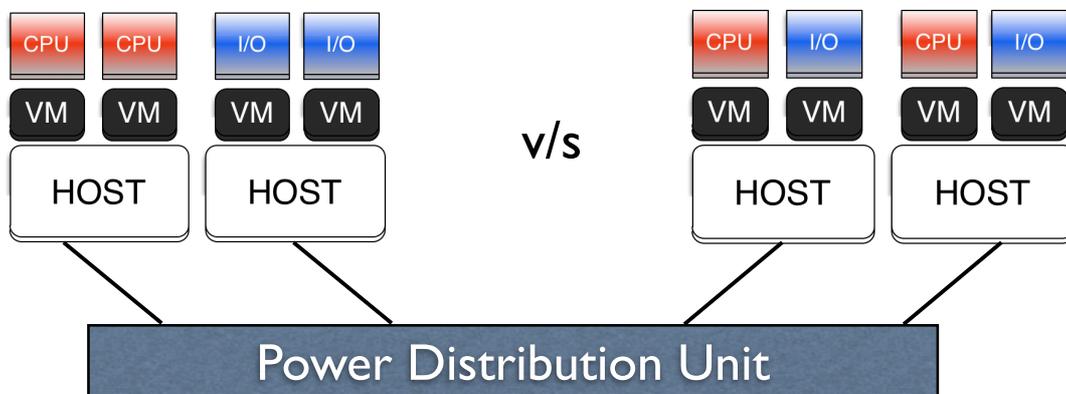


Figure 4.2: Diagram of workloads inside VMs and their allocations

4.2.2.2 Results

Mix	Host A	Host B	Total
Co-located	190.5 ± 0.7 W/h	115.5 ± 0.7 W/h	306 W/h
Split	159 ± 0 W/h	161.5 ± 0.7 W/h	320.5 W/h

Table 4.3: Mean energy usage results from the Virtualization Effect experiment (3 runs)

The mean watt-hour values for this experiment are shown in the table above. The experiment was repeated three times and on each occasion the individually reported figures varied by no more than a single watt-hour from the mean, with a standard deviation of 0.7 W/h.

The same statistical analysis from the previous experiment was applied, giving a p-value of 0.001012 from the T-Test so that it is very unlikely, about a one in one thousand chance, these results could have arisen from a sampling variability.

It is clear from the table above that the two mixes produced different energy usage totals. The only variable that was changed was the workload mix, so the mix has had an effect. The optimal workload mix (co-located) required 95.3% of the energy required for the non-optimal mix (split). Introducing a virtualization layer has an effect on the energy used by the system, and in particular seems to use less energy. The virtualization layer introduces additional overhead to the processes so reducing throughput. The fact that each task does not run at its fastest level where it would use more energy, could account for this.

As virtualization is a key characteristic of Private cloud computing platforms, these results show that workload mixing can be used to reduce the energy consumption of such systems.

4.2.3 Application performance vs. Energy usage experiment

The third experiment in the series used the *Phoronix* benchmark suite [92] in place of the synthetic workload generating applications. *Phoronix* measures real world applications such as *gzip* and provides scores for their

performances. This allowed us to compare the effect of mixes on both power and performance. The *Phoronix* test suite was chosen because it provides a variety of benchmarking tools is open source and provides built-in uploading of results to *openbenchmarking.org* for easy comparison of different experiment runs.

The aim of this experiment was firstly to investigate if the previous conclusions regarding workload mixes remain applicable when real-world applications are used, and secondly, to determine the effect workload mixes have on the performance of those applications, running on virtual machines.

4.2.3.1 Experimental design

In this experiment there are two types of task, one that is CPU bound and one that is Disk (DISK) bound. The CPU bound task is a benchmark of the *gzip* application – a standard Linux application that can compress files in memory. The I/O Bound task is a benchmark called *aio-stress* is an asynchronous I/O benchmark created by SuSE. It uses a single thread to consistently read and write a 1024MB test file to and from the hard disk.

As before, we use two VMs with half of the host server’s physical resources to maintain consistency with the previous experiments. The underlying hardware and monitoring solutions remain the same.

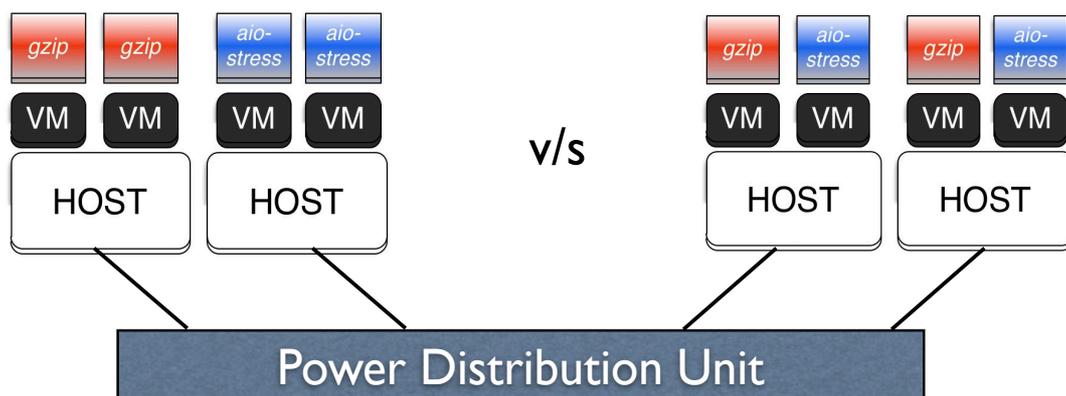


Figure 4.3: Diagram of real-world applications in two different workload mixes

As in the previous experiment, a square wave usage pattern was adopted.

4.2.3.2 Energy results

Mix	Host A	Host B	Total
Co-located	139 ± 0.8 W/h	108 ± 1.2 W/h	247 ± 1.9 W/h
Split	127 ± 0.9 W/h	125 ± 0.5 W/h	252 ± 1.2 W/h

Table 4.4: Mean energy usage results from Performance v Energy experiment (3 runs)

The mean energy draw values for this experiment are shown in the table above. The total energy used values are outwith one standard deviation, the p-value of these results from a t-test is 0.064, a low value, so we conclude that it is unlikely these results could have arisen from a sampling variability. We therefore adopt the conclusion that the workload mix affected the energy consumption. The energy usage difference was small, but statistically significant.

It is clear from the table above that the two mixes produced different energy usage totals. The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 98% of the power required for the non-optimal mix (split). The difference in energy usage between this result and the previous one suggests that that these benchmark tasks required less energy to work than the generic workload tasks that were previously used.

4.2.3.3 Performance results

Mix	<i>gzip</i> (CPU)	<i>aio-stress</i> (DISK)
Co-located	18.73 ± 0.03 s	97.41 ± 29.38 MB/s
Split	18.83 ± 0.03 s	32.20 ± 1.14 MB/s

Table 4.5: Performance results from Performance v Energy experiment (3 runs)

gzip scores are measured as seconds to compress a 2GB binary file. Therefore a lower score is better. *aio-stress* is measured as I/O throughput per second meaning a higher score is better. The mean performance scores for this experiment are shown in the table above. Analysis revealed a p-value of 0.041

for *aio-stress* and 0.011 for *gzip*, so we conclude that the workload mix also affected the application performance scores.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the energy required to power the servers and hold or increase the performance. The split mix required more energy and produced a slower *gzip* score and less *aio-stress* throughput. CPU performance was similar in both experiments but disk throughput on the co-located mix was over 3x the disk throughput of the split mix.

This is an important result as it demonstrates that energy usage can be improved without perceptible loss of performance, and, in some cases, performance may also be improved, particularly in this case DISK performance.

4.2.4 Additional performance vs. Energy consumption experiments

Subsequent experiments were derived to examine the interplay of different application types in a similar scenario to the “Application Performance vs. Energy Consumption” experiment detailed in this chapter. They have the same aims and requirements.

In this experiment, there are different types of tasks: CPU bound, Memory bound, network bound and I/O (DISK) bound (CPU, MEM, NET and DISK).

- The CPU bound task is a benchmark of the *gzip* application – a standard Linux application that can compress files in memory.
- The MEM bound task is a benchmark called *stream* that evaluates the system memory performance.
- The NET bound task evaluates the system network performance by transmitting and receiving a large file over the to and from the test computer to evaluate transmit and receive speed.
- The I/O bound task is a benchmark called *aio-stress* is an asynchronous I/O benchmark created by SuSE. It uses a single thread to consistently read and write a 1024MB test file to and from the hard disk.

As in the previously described experiment, we use two VMs with half of the host server’s physical resources (4 virtual CPU cores and 8GB of RAM per VM) to maintain consistency with the previous experiments. The underlying hardware and monitoring solutions remain the same.

These experiments involve four tasks (two of each type) executed concurrently in single virtual machines for one hour. The results of these experiments will now be described in detail:

4.2.4.1 CPU & MEM

4.2.4.1.1 Experimental design

In the co-located mix, co-located virtual machines are given tasks of the same type – CPU intensive tasks are located in VMs on Host A and MEM intensive tasks are located in VMs on Host B.

In the split mix, the similar tasks are split between both hosts. Host A and B will now have VMs running one CPU task and one MEM task.

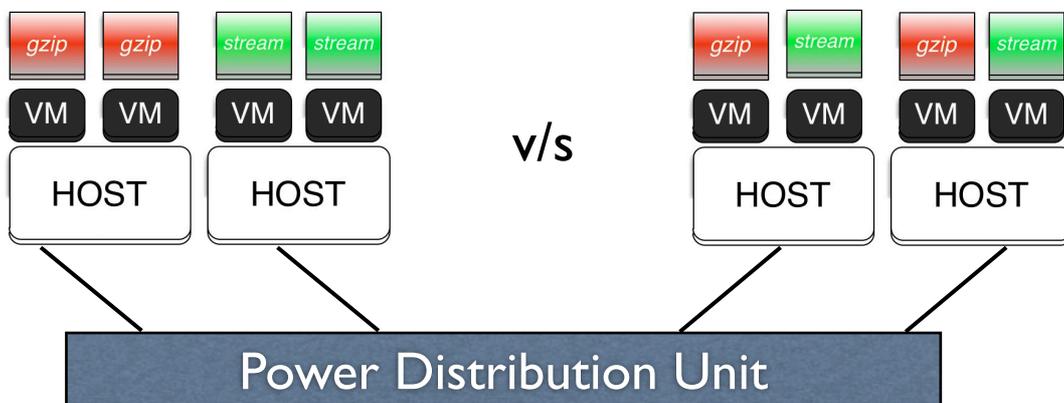


Figure 4.4: Diagram of Mixes A and B (CPU and MEM applications)

4.2.4.1.2 Power Results

Mix	Host A	Host B	Total
Co-located	139 ± 8.0 W	103 ± 16.0 W	243 ± 17.8 W
Split	130 ± 12.9 W	129 ± 12.7 W	259 ± 19.4 W

Table 4.6: Mean power draw from CPU v MEM experiment

The mean power draw values for this experiment are shown in table 4.6. It is clear from the table above that the two mixes produced different power usage. The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 93.8% of the energy required for the non-optimal mix (split). P-value for the difference in power results is 8.6×10^{-84} , so we can conclude that the result did not arise as the result of sampling variability.

4.2.4.1.3 Performance results

Mix	<i>gzip</i>	<i>stream</i>
Co-located	19.09 ± 0.15 s	7573 ± 747 MB/s
Split	18.91 ± 0.18 s	6910 ± 551 MB/s

Table 4.7: Performance results from CPU v MEM

gzip scores are measured as seconds to compress a 2GB binary file. Therefore, a lower score is better. *stream* is measured as memory throughput per second, meaning a higher score is better. The mean performance scores for this experiment are shown in table 4.7.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the power required for the servers and increase memory performance but slightly degrade the CPU performance (by 1% on average for this application). The split mix required more energy, produced a marginally faster *gzip* score and less *stream* throughput. In this particular scenario, significant power savings were achieved with only 1% decrease in CPU performance, likely to be imperceptible at the user level. P-

values for the *gzip* results are 2.66×10^{-10} and 4.57×10^{-9} for *aio-stress*, we can therefore conclude that the results did not arise from sampling variability.

4.2.4.2 CPU&NET

4.2.4.2.1 Experimental design

In the co-located mix, co-located virtual machines are given tasks of the same type – NET intensive tasks are located in VMs on Host A and CPU intensive tasks are located in VMs on Host B.

In the split mix, the similar tasks are split between both hosts. Host A and B will now have VMs running one CPU task and one NET task.

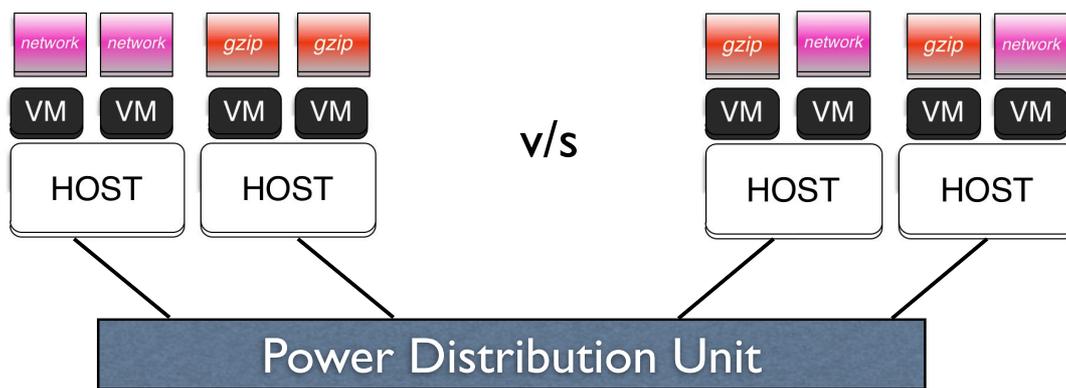


Figure 4.5: Diagram of Mixes A and B (NET and CPU applications)

4.2.4.2.2 Power results

Mix	Host A	Host B	Total
Co-located	151 ± 7.6 W	137 ± 7.5 W	288 ± 11.1 W
Split	145 ± 8.8 W	147 ± 9.4 W	292 ± 13.4 W

Table 4.8: Mean power draw from NET v CPU experiment

The mean watt-hour values for this experiment are shown in the table above. It is clear from the table above that the two mixes produced different energy usage totals. The only variable that was changed was the workload mix, so the

mix must have had an effect.

The optimal workload mix (co-located) required 98.6% of the power required for the non-optimal mix (split). The small difference in power required for each mix would suggest that the CPU and NET tasks both use a similar amount of energy and are not as affected by the workload mix pairings as other tasks. It would seem that CPU and NET tasks influence the system in the same way and the different mixes do not have much of an effect.

4.2.4.2.3 Performance results

Mix	<i>network</i>	<i>gzip</i>
Co-located	41.24 ± 0.92 s	18.71 ± 0.07 s
Split	41.59 ± 1.27 s	18.87 ± 0.21 s

Table 4.9: Performance results from NET v CPU experiment

gzip scores are measured as seconds to compress a 2GB binary file. Therefore a lower score is better. *network* is measured as the time taken to transmit and received a file through the network so a lower score is better. The mean performance scores for this experiment are shown in the table above. P-value for *gzip* in this experiment is 1.85×10^{-08} and for *network* it is 0.147. The p-value and standard deviation would suggest that the difference between the *network* values in each experiment is not statistically significant.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the energy required to power the servers and marginally improve the CPU performance. Both the CPU and NET intensive applications showed small increases in performance when compared with the alternative mix, although the NET results are not statistically significant. This is one of the results that shows it is possible to reduce the energy consumption of a server without reducing the performance of the applications running on that server.

Similarities in the performance figures for CPU and NET jobs, despite different workload mix placements, suggests that these types of jobs place

similar loads on the hardware subcomponents.

4.2.4.3 MEM & NET

4.2.4.3.1 Experimental design

In the co-located mix, co-located virtual machines are given tasks of the same type – MEM intensive tasks are located in VMs on Host A and NET intensive tasks are located in VMs on Host B.

In the split mix the similar tasks are split between both hosts. Host A and B will now have VMs running one MEM task and one NET task.

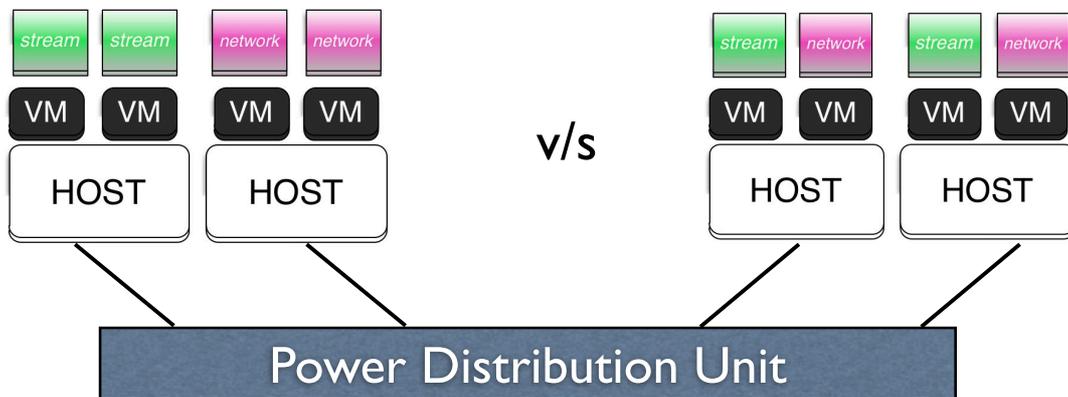


Figure 4.6: Diagram of Mixes A and B (MEM and NET applications)

4.2.4.3.2 Power results

Mix	Host A	Host B	Total
Co-located	103 ± 14.6 W	154 ± 7.4 W	257 ± 16.3 W
Split	134 ± 10.8 W	139 ± 12.3 W	273 ± 17.2 W

Table 4.10: Mean power draw from MEM v NET experiment

The mean power draw values for this experiment are shown in the table above. P-value for the total power results is 2.36×10^{-106} . The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 94.1% of the energy required for

the non-optimal mix (split).

4.2.4.3.3 Performance results

Mix	<i>stream</i>	<i>network</i>
Co-located	8219 ± 541 MB/s	42.14 ± 2.11 s
Split	7625 ± 562 MB/s	44.59 ± 1.29 s

Table 4.11: Mean performance results from MEM v NET experiment

stream is measured as memory throughput per second meaning a higher score is better. *network* is measured as the time taken to transmit and received a file through the network so a lower score is better. The mean performance scores for this experiment are shown in the table above. P-value for the difference in *stream* performance for each mix is 1.75×10^{-11} , and for *network* it is 1.45×10^{-5} , so we adopt the explanation that these differences are unlikely to have arisen as the result of sampling variability.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the energy required to power the servers and at the same time improve the application performance. Both the MEM and NET intensive applications showed increases in performance when compared with the split mix.

4.2.4.4 DISK & MEM

4.2.4.4.1 Experimental design

In the co-located mix, co-located virtual machines are given tasks of the same type – DISK intensive tasks are located in VMs on Host A and MEM intensive tasks are located in VMs on Host B.

In the split mix the similar tasks are split between both hosts. Host A and B will now have VMs running one DISK task and one MEM task.

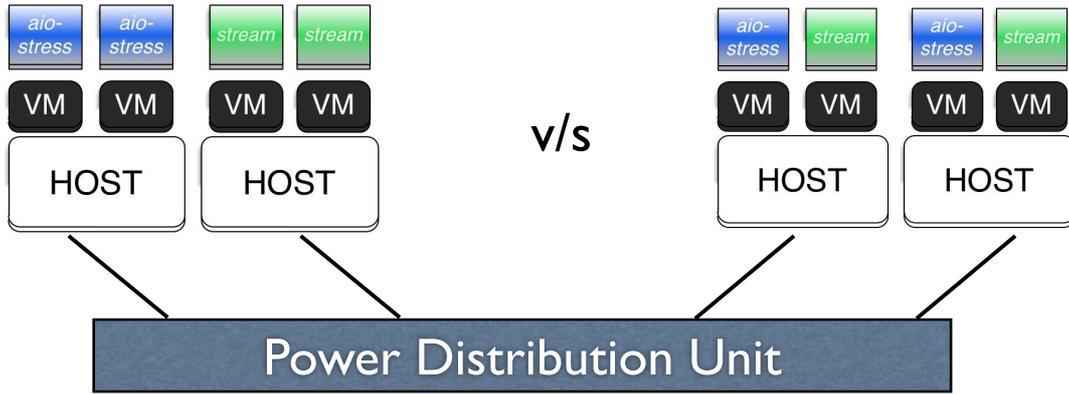


Figure 4.7: Diagram of Mixes A and B (DISK and MEM applications)

4.2.4.4.2 Energy results

Mix	Host A	Host B	Total
Co-located	112 ± 6.9 W	102 ± 14.9 W	214 ± 16.5 W
Split	117 ± 10.0 W	115 ± 12.1 W	232 ± 15.9 W

Table 4.12: Power results from DISK v MEM experiment

The mean power draw values for this experiment are shown in the table above. It is clear from the table above that the two mixes produced different energy usage totals. P-value for the total power result is 9.57×10^{-123} . The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 92.2% of the energy required for the non-optimal mix (split).

4.2.4.4.3 Performance results

Mix	<i>aio-stress</i>	<i>stream</i>
Co-located	98.51 ± 58.3 MB/s	7657 ± 770 MB/s
Split	187.42 ± 34.9 MB/s	7226 ± 474 MB/s

Table 4.13: Performance results from DISK v MEM experiment

aio-stress is measured as I/O throughput per second meaning a higher score is better. *stream* is measured as memory throughput per second meaning a

higher score is better. The mean performance scores for this experiment are shown in the table above. P-value for *aio-stress* is 0.004 and for *stream* it is 0.079. The latter value, and given the standard deviation values for *stream* results, should indicate that there is no statistical difference between the *stream* values for this workload.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the energy required to power the servers and maintain the MEM application performance. However, DISK performance for the co-located mix was around 52% of the throughput of the split mix.

The DISK application performance is curious, as in this case the DISK performance was better when the tasks were co-located with MEM tasks. The next experiment will examine the final possible pairing for DISK, with NET tasks, so we will be able to draw conclusions once it has been described.

4.2.4.5 *NET & DISK*

4.2.4.5.1 Experimental design

In the co-located mix, co-located virtual machines are given tasks of the same type – NET intensive tasks are located in VMs on Host A and DISK intensive tasks are located in VMs on Host B.

In the split mix the similar tasks are split between both hosts. Host A and B will now have VMs running one DISK task and one NET task.

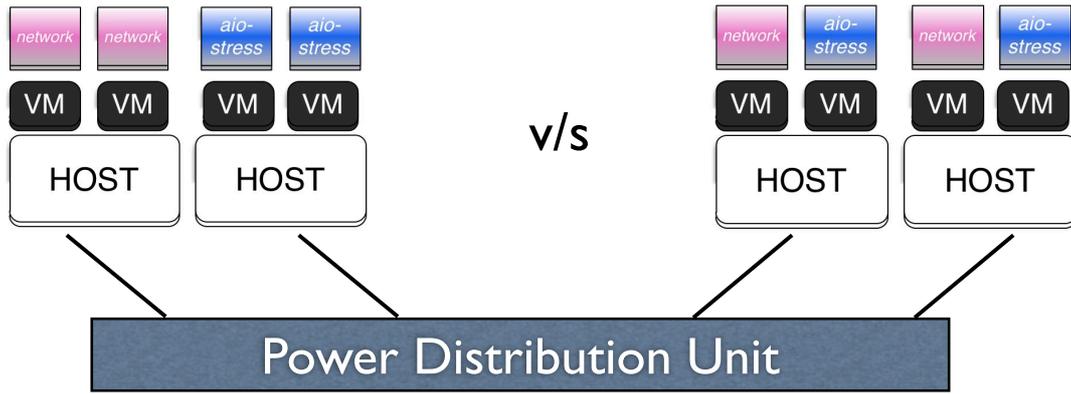


Figure 4.8: Diagram of Mixes A and B (NET and DISK applications)

4.2.4.5.2 Power results

Mix	Host A	Host B	Total
Co-located	136 ± 14.2 W	106 ± 9.2 W	242 ± 16.9 W
Split	135 ± 8.6 W	139 ± 10.6 W	274 ± 14.2 W

Table 4.14: Mean power draw from NET v DISK experiment

The mean power draw values for this experiment are shown in table 4.14. It is clear from the table above that the two mixes produced different energy usage totals. P-value for the power totals was 3.82×10^{-288} . The only variable that was changed was the workload mix, so the mix must have had an effect. The optimal workload mix (co-located) required 88.9% of the power required for the non-optimal mix (split). This lower amount of power used would suggest that DISK and NET tasks respond well to the workload mix configurations and would benefit in future deployments for being arranged in the manner of Mix A.

4.2.4.5.3 Performance results

Mix	<i>network</i>	<i>aio-stress</i>
Co-located	42.42 ± 1.38 s	103.63 ± 64.2 MB/s
Split	43.01 ± 0.89 s	186.62 ± 54.1 MB/s

Table 4.15: Performance results from NET v DISK experiment

aio-stress is measured as I/O throughput per second meaning a higher score is better. *network* is measured as the time taken to transmit and received a file through the network so a lower score is better. The mean performance scores for this experiment are shown in table 4.15. P-value for the *network* comparison is 0.011, and for *aio-stress* it is 0.009.

For this particular workload, on these particular servers, one workload mix (co-located) was demonstrated to reduce the energy required to power the servers and improve the NET application performance. Disk performance was impacted, suggesting that DISK tasks do not always benefit from co-location.

We can now draw conclusions about the performance of DISK based jobs:

- In this case once again, DISK performed better in the split, higher energy consuming workload mix. It appears that when DISK jobs are co-located together on this hardware configuration the *aio-stress* throughput is approximately 100MB/s.
- When those tasks are split, their performance depends entirely on the task they are paired with. Non-CPU based tasks, such as MEM and NET cause the DISK performance to increase to more than 150MB/s.
- If paired with a CPU job, the DISK performance can degrade to less than 50MB/s.
- Regardless of performance, co-locating DISK jobs of the same type always gives the lowest energy usage on this hardware. A VM allocation strategy that aims to reduce energy will therefore focus on pairing DISK jobs together as default.

The disk performance discussed here is not a general result; it is specific to the hardware and software configurations used in this set of experiments. The degraded I/O performance when paired with CPU tasks is most likely due to the high amount of CPU utilization blocking the I/O operations and degrading performance.

4.2.5 VM sizes

This fourth experiment deviates from the standard pattern of virtual machine sizes to investigate the effect of different VM configurations on application performance and energy consumption.

Three virtual machine sizes are used:

- m1.small - 1 CPU core and 1GB of RAM
- m1.medium - 2 CPU cores and 2GB of RAM
- m1.large - 4 CPU cores and 8GB of RAM

These sizes are based on the virtual machine sizes offered by the St Andrews cloud computing co-laboratory's (StACC) OpenStack private cloud.

4.2.5.1 Experimental design

This experiment was conducted as a repeat of a co-located mix similar to the previous experiment, "*Application performance vs. Energy consumption*", with varying virtual machine sizes employed.

As before, we use two types of phoenix benchmark test suite applications, *gzip* and *aio-stress*, to give examples of different workloads. The underlying hardware and monitoring solutions remain the same as the previous experiments.

The experiment involves four tasks (two of each type) executed concurrently in single virtual machines for one hour. Only one mix is used; co-located virtual machines are given tasks of the same type – CPU intensive on Host A and DISK intensive tasks on Host B. This layout is the same for each virtual

machine size.

4.2.5.2 Energy results

VM Size	Host A (<i>aio-stress</i>)	Host B (<i>gzip</i>)	Total
Small	109.7 ± 0.56 W/h	139.7 ± 1.53 W/h	249.3 ± 1.53 W/h
Medium	110 ± 1 W/h	138 ± 1 W/h	248 ± 1 W/h
Large	108.3 ± 1.53 W/h	139 ± 0.58 W/h	247.3 ± 1.53 W/h

Table 4.16: Energy results from VM Sizes experiment

The mean watt-hour values for this experiment are shown in the table above. The size of the virtual machine had a marginal impact on the amount of energy used and instead the type of work performed dominated the power profile. The differences between the energy used values for different VM sizes are not statistically significant. This would suggest that configuration of each VM has little or no effect on the amount of energy consumed. A larger VM will use the same amount of energy as a smaller VM if the same load is applied.

4.2.5.3 Performance results

VM Size	<i>aio-stress</i>	<i>gzip</i>
Small	17.55 ± 0.68 MB/s	27.64 ± 0.29 s
Medium	18.20 ± 3.25 MB/s	27.45 ± 0.09 s
Large	97.41 ± 35.99 MB/s	18.75 ± 0.03 s

Table 4.17: Performance results from VM Sizes experiment

The mean performance scores for this experiment are shown in the table above. p-value was 7.651×10^{-07} for *gzip* and 0.018 for *aio-stress*, so we conclude that the virtual machine sizes affected the application performance scores. With the p-values and differences between performance results falling more than one standard deviation away, we can conclude that the results were statistically significant.

These results show that energy usage is almost constant across all virtual machine sizes, yet performance results vary considerably, with larger VMs giving increased performance. CPU performance on large machines was 1.5x the medium instances and disk throughput was over 5x.

This would suggest that constraining VM size for energy-saving purposes in a private cloud is ineffective, especially given the large detriment to performance that is experienced. In a situation where a larger virtual machine size is chosen the superior performance may mean a finite piece of work is completed quicker, thereby reducing the energy consumption in a different manner. The increased performance of a larger VM comes at no additional energy cost.

This result also raises the question of higher prices for larger VMs on a public cloud. If the larger VM uses almost the same amount of energy as a smaller configuration, then public cloud providers might see an increase in their profit margins as no additional energy costs are incurred.

4.2.6 Mixes vs. bin-packing

For the fifth and final experiment, we compare the operation of a workload mix where four virtual machines containing four tasks in a spread configuration are compared against a stacked bin-packing configuration where the virtual machines are placed on a single server.

We take the results of the m1.medium virtual size from the previous experiment and compare those with the operation of a bin-packing configuration of four m1.medium VMs on a single host.

4.2.6.1 Experimental design

This experiment utilized four m1.medium VMs on a single compute node and compared the results to that from the previous experiment.

As before, we use two types of *Phoronix* benchmark test suite applications, *gzip* and *aio-stress* to give examples of different workloads. The underlying hardware and monitoring solutions remain the same as for previous

experiments.

The experiment involves four tasks (two of each type) executed concurrently in single virtual machines for one hour. Only one mix is used, four co-located virtual machines on a single host. For the energy score, the base load of the unused compute node is added to the totals.

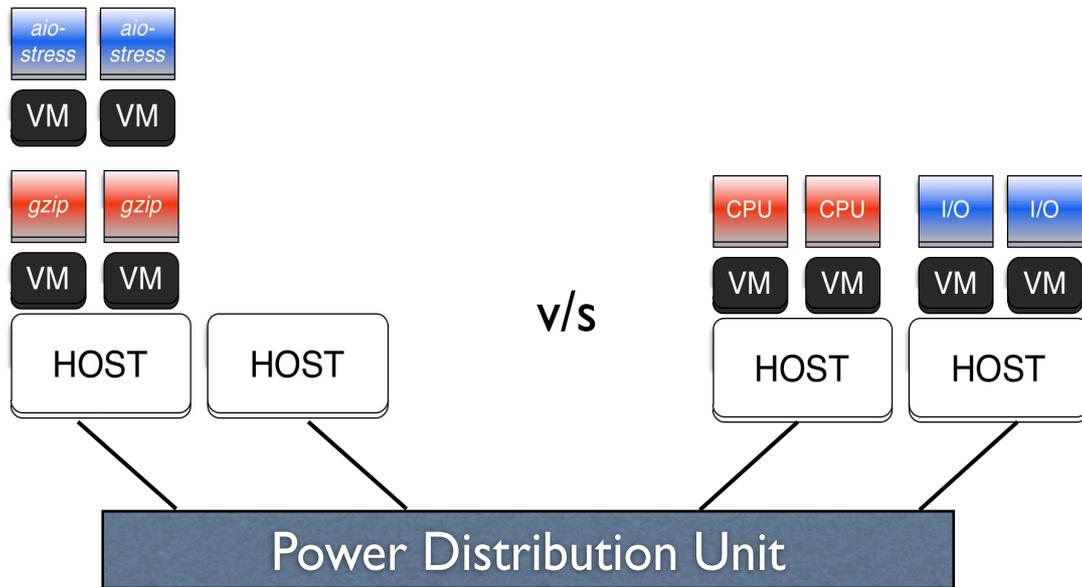


Figure 4.9: Diagram of stacked, bin-packed workload mix against a spread configuration
As in the previous experiment, a square wave usage pattern was adopted.

4.2.6.2 Energy results

Allocation	Host A	Host B	Total
Bin-packed	124 ± 1 W/h	74.67 ± 0.58 W/h	198.67 ± 1.15 W/h
Workload Mix	138 ± 1 W/h	110 ± 1 W/h	248 ± 1 W/h

Table 4.18: Energy results from Bin-packing v Workload Mixes experiment

The bin-packing layout used, on average, 19.9% less energy than the workload mix alternative. Results over the experiment were consistent, with a standard deviation of at most 1.15 W/h.

Analysis of these results revealed a p-value of 0.018, so we conclude that it is

unlikely that these results could have arisen from a sampling variability. We therefore adopt the conclusion that the bin-packing layout had a different affect on energy consumption than the workload mix layout.

4.2.6.3 Performance results

Allocation	<i>aio-stress</i>	<i>gzip</i>
Bin-packed	6.55 ± 2.19 MB/s	45.47 ± 2.4 s
Workload Mix	18.20 ± 2.65 MB/s	27.45 ± 0.08 s

Table 4.19: Performance results from Bin-packing v Workload Mixes experiment

The mean performance scores for this experiment are shown in the table above. Analysis revealed a p-value of 0.02 for *gzip* and 0.003 for *aio-stress*, so we conclude that the bin-packing layout affected the application performance scores compared to the workload mix effect.

The bin-packing stacked virtual machine layout impacts application performance in a negative way but requires less energy to execute. It is intuitive that a host with stacked VMs contending for resources will result in worse application performance than a configuration where the virtual machines are spread out.

The workload mix disk throughput was 2.7x the bin-pack performance and had 1.65x the CPU performance. However, the workload mix scenario required approximately 1.25x the energy of the bin-pack over the same time period.

Bin packing requires less energy to run than a spread layout. If workload mixes are to be employed, there is no question that the best energy usage pattern can be achieved using a bin-packing scheduling algorithm, but at significant detriment to system performance and dependability. By packing VMs into the least number of VMs possible, dependability can be impacted if those VMs are copies of the same task. Administrators may often launch multiple VMs running the same task to either balance workload between them or to have backups should the primary task fail. When bin packing is used, there is a higher chance that these redundant VMs will be co-located on the

same physical hosts, resulting in a greater loss of dependability if a single host should fail.

4.3 Discussion & conclusion

This chapter demonstrated that workload mixes affect application performance and system energy. Workload mix scheduling based on the characteristics of applications is a spread approach to assigning work that takes into consideration how virtual machine placement will effect both energy and performance.

Traditional approaches such as bin-packing potentially use less energy if servers can be switched off and are not concerned with dependability, over-working of hardware or reaction time. If such attributes are important to system administrators, workload mixes can help to optimize workload layout so that energy usage is minimized while still meeting these requirements.

The next chapter discusses the development of a workload placement strategy for private cloud systems that exploits VM workload characteristics to mix jobs in such a way that system energy usage is minimized while maintaining VM performance and application dependability.

The strategy is implemented in such a way that the workload mixes can be adapted depending on the read-world application performance, energy usage and changing IT policies. This allows it to be deployed on heterogeneous hardware and take advantage of gains that may be found when workload allocations are studied, such as a new workload mix that will enhance system performance.

This work is presented as a real world validation of a theoretical framework that could be adapted in the future. The actual effects of various mixes on different hardware can be investigated if different hardware is used, but in future work this could be done automatically for heterogeneous compute nodes.

It is conceivable that different hardware servers could have widely different

results from those presented in this chapter. However, within the practicalities of our environment we believe the conclusions to this work are solid. It is logical that CPU task performance is very similar in different workload configurations, even when those tasks are co-located, due to the advances in multi-tasking technology in modern CPU chips.

The experiment results could be affected if there are unknown bugs or issues in the *CloudMonitor* application or the software used to run these experiments. Similarly, any variability or failures in the PDU would also impact validity. The results presented here are done so under the assumption that such issues were not significant in their effect on the data.

The DISK performance is also worth highlighting, as our investigations lead us to believe that the perhaps counter-intuitive results of performance being boosted in co-located DISK tasks (when compared to when DISK tasks are paired with CPU intensives tasks) are the result of attempted optimizations by the KVM hypervisor. As such these results are valid for the future development of a workload placement strategy on a private cloud that uses KVM.

5 Implementation of OpenStack *Allocator*

This chapter discusses how workload mixes have been implemented as the *Allocator* software on the private cloud computing platform, OpenStack [93]. The strategy adopted in this thesis aims to exploit the characteristics of workload mixes to intelligently allocate new virtual machine instances within the private cloud in a manner that optimizes the amount of energy used while maintaining application performance. A workload mix is the arrangement of a finite set of jobs on a finite set of computational hardware units. The mix of those jobs is made up of their placement decisions and the characteristics of those jobs.

The implementation of this VM placement strategy using workload mixes is one of the major contributions to this thesis and was designed to allow an organization to reduce their energy bills and carbon footprint while retaining the necessary computational power to complete their work and without change to system management policies on powering machines up and down.

The strategy was implemented on OpenStack as a software system called *Allocator*. The software evaluates VM placement choices with respect to the lessons learned from the experiments in Chapter 4 and advises the Openstack system accordingly.

Administrators can configure the system by defining workload mix rules that the implementation will adhere to. These rules are defined using a simple English-based rules language. The workload mix rules that are executed are completely customizable, allowing system administrators to tailor the allocation policy to their IT policies. This rule-based approach helps to future-proof the implementation by allowing new policies to be adopted if a new workload mix is found to be more efficient or if the system administrator is required to adapt the system to new IT policies.

Users interact with the new system in almost the same way as a non-modified version of OpenStack. They are asked to add one additional command flag when submitting a new VM instance request. This command "*vm_type*" specifies the dominant computational resource the VM and its internal application will use. These are drawn from the four major parts of a

computing system: central processing unit (CPU), system memory (MEM), network (NET), and hard drive (DISK). Once the command has been specified the user will interact with the system and their VM as they normally would. A user guide is included as Appendix A of this thesis.

Assigning a VM a single type allows the custom code to apply the lessons from Chapter 4. Observations and discussions with cloud users suggest that users of private cloud tend to launch one virtual machine to perform one task, and launch additional VMs as new tasks arise. We have assumed this model of VM allocation in our work.

The system relies upon the users to cooperate with the additional command and provide accurate descriptions of the workloads that will be executed. The system was designed from the beginning to be simple and easy to use for end users, eliminating any temptation to bypass the system and therefore ensuring a high level of user compliance when it is deployed.

If users are unsure about their application profile they can run it once alongside the *CloudMonitor* software detailed in Chapter 3. This software will provide a detailed breakdown of the applications use of major resources allowing an informed user to accurately label new instance requests for this application.

The software of this implementation of workload mixing was developed using the agile method *Scrum* over a 6-month development period, with regular sprints followed by demonstrations of the emerging functionality. The system has a minimal front-end interface, so additional tools and tracking facilities were developed to monitor and display system decisions and outcomes.

The implementation and assorted tools consists of around 5,000 lines of program code, mostly Python, and is available online under the open-source Apache license. The code is available on Github.com²⁶, which was used throughout development along with the *git* source code management and revision tool to maintain our project files.

²⁶ <http://github.com/jws7/allocator>

This chapter describes the design and implementation of the workload mix implementation, including a description of how the new custom code fits in with the OpenStack project. This chapter starts with a description of the OpenStack project and system architecture (Section 5.1) then moves on to describe the architecture of the *Allocator* (Section 5.2), the rules language that is used by the *Filter* module (5.3), the *Filter* module itself (Section 5.4), *Weigher* module (Section 5.5), data storage and decision tracking (Section 5.6), how VM terminations are tracked (Section 5.7) before the chapter concludes with a description (Section 5.8) of how the code is deployed and customized within OpenStack itself.

5.1 The OpenStack project and system architecture

NASA and Rackspace Hosting founded the OpenStack cloud computing project in July 2010 in a joint venture as a combination of their Nebula and Cloud Files platforms. Today, it is an open-source project featuring over 400 developers from around the world. It has grown from 30,000 lines of code in the initial release, codenamed *Austin*, to over 600,000 in the latest 2013 release *Grizzly*.

The project is intended as an *Infrastructure-as-a-Service* platform, allowing organizations and individuals to deploy cloud computing services without specialized hardware. It is similar in scope to other open-source projects like Eucalyptus, but has now become the *de facto* standard for those who wish to deploy an open-source cloud computing solution^{27 28}.

In the St Andrew cloud computing co-laboratory (StACC) an OpenStack installation provides a test bed for academic researchers who wish to experiment with cloud technologies. For this reason, along with its emerging popularity and open nature, OpenStack was chosen as an experimental

²⁷ <http://www.infoworld.com/d/cloud-computing/OpenStack-gains-momentum-vendors-give-grizzly-bear-hug-216458>

²⁸ <http://www.OpenStack.org/blog/2012/08/OpenStack-won-unprecedented-popularity-in-asiapacific/>

platform to demonstrate the effectiveness of our workload mixing strategy.

OpenStack has a complicated architecture, as can be seen from Figure 5.1. For the purposes of developing a custom energy-efficient VM allocation strategy, we are concerned with the “nova-schedule” part of the above architecture diagram.

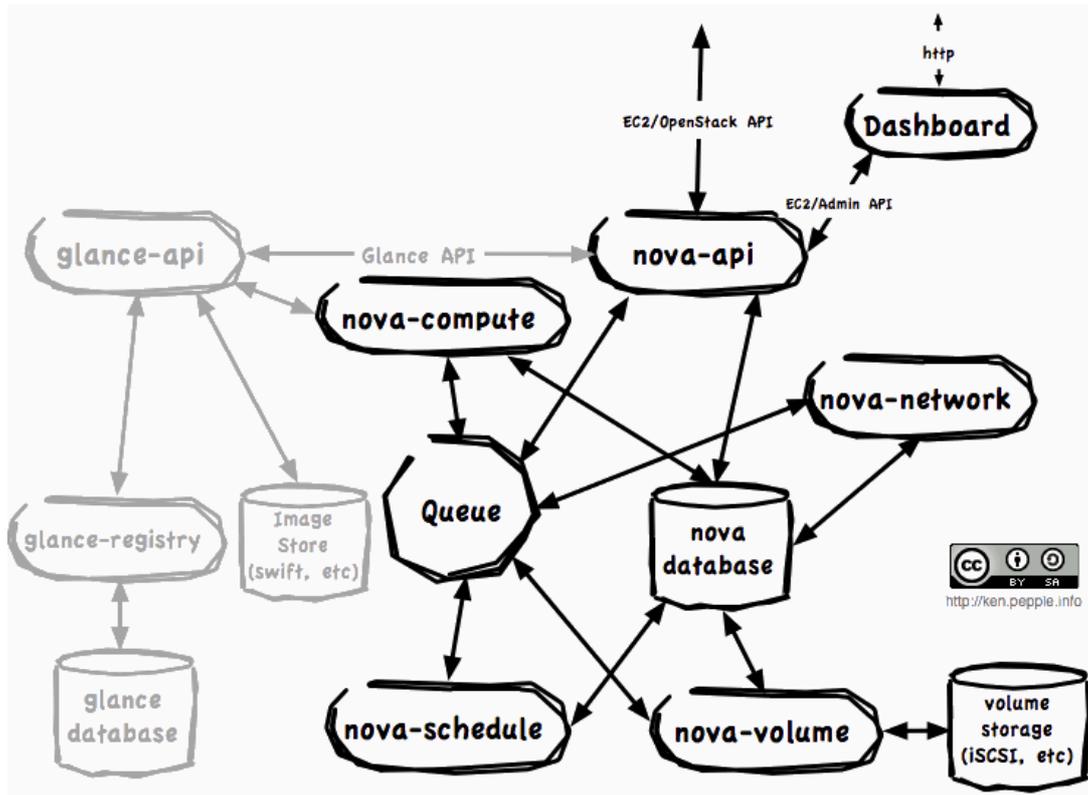


Figure 5.1: OpenStack project architecture diagram by Ken Pepple²⁹

nova is the core part of the OpenStack project that is used to control and manage virtual machine creation, allocation and termination. It communicates with other parts of OpenStack through a messaging system and scales out horizontally across all of the compute nodes without any specialized or proprietary hardware or software requirements. Each node in the system has a “*nova-controller*” installed, a daemon that reports the node’s current state and operates the virtual machines by interacting with the node hypervisor.

²⁹<http://ken.pepple.info/OpenStack/2011/04/22/OpenStack-nova-architecture/>

nova can work with many different hypervisor technologies such as KVM and XenServer; in the StACC OpenStack installation, the KVM hypervisor has been chosen by our system administrator and will be the one used for this implementation. This choice of KVM was also used for the experiments outlined in Chapter 4, so lessons learned there are directly translatable for the development of this implementation. The specific hypervisor in use is abstracted away from any of the *nova* code that was modified in the course of this development.

The part of the *nova* code which is concerned with VM scheduling is the *FilterScheduler*, a placement decision engine which allows different *Filters* and *Weighers* to be used to influence how jobs are allocated to VMs.

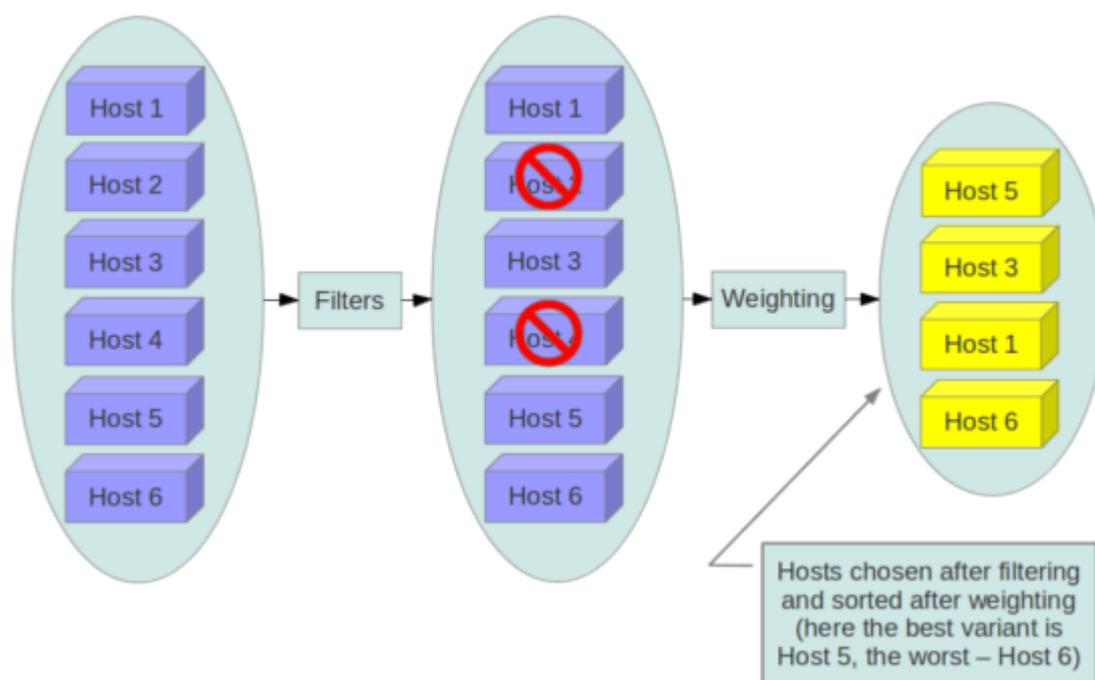


Figure 5.2: Example of Filter Scheduler from OpenStack.org

Figure 5.2 is an example of how it works. Given a choice of a number of hosts in which to place a new VM instance, the *FilterScheduler* will first load the specified *Filter* code to remove any hosts that are unsuitable. The remaining host are then “weighed” and assigned a score. The hosts are ranked by their scores and the *best* one chosen as the host for the new VM instance. To create a *Allocator* strategy it is necessary to write a custom *Filter* and *Weigher*.

“Scheduler Hints” is a *FilterScheduler* mechanism within the *nova* API to attach a key-value dictionary to the *Filter* and *Weigher* modules. The dictionary can be used to pass commands to both the *Filter* and the *Weigher* code. In the OpenStack documentation, the example used is to request that a new VM instance is co-located with an existing virtual machine. This is achieved by passing the hint “same_host” to the *nova* boot command:

```
nova boot ... --hint same_host=[<instance-uuid>]
```

For this to work, the *SameHostFilter* must be engaged in the OpenStack configuration. If so, the *Filter* will then remove any hosts from the possible nodes that do not have a VM with the passed instance *uuid*. The scheduler may choose to ignore hints if they are not compatible with the current situation or if a *Filter* is used that does not require the hint provided.

The *Allocator* implementation described within this chapter uses the hint *vm_type* to specify the type of work the instance will be conducting. For example, for a CPU bound instance, the user should request an instance using a command similar to this:

```
nova boot ... --hint vm_type=CPU
```

This will notify the custom *Filter* and *Weigher* modules that the user wishes to use this new instance for CPU intensive work and will attempt to allocate accordingly. Other appropriate types are DISK, MEM and NET.

The custom *Allocator* software fits into the OpenStack architecture by implementing new *Filter* and *Weigher* modules. The custom *Filter* is intended to remove inappropriate hosts for new VMs based on the *vm_type* hint and the custom workload mix rules that are configured to be applied.

The *Allocator Weigher* module then uses the workload mix knowledge from Chapter 4 to score each potential host to decide on the most appropriate location for the new VM.

5.2 *Allocator* architecture

Python is one of the most popular programming languages, used by an estimated 1 million developers worldwide [94]. There are many open-source projects that have been written in Python, including OpenStack. This popularity means that Python is an ideal modern programming language thanks to its developer community, extensive libraries and many guides and tutorials on the World Wide Web. For example, Python can be used with any modern database system and most of the widely used databases such as MySQL have open-source, well-maintained libraries for abstracting over the more common database interaction commands.

The *Allocator* code was written in Python to allow it to easily fit in with the OpenStack code without any language translation interfaces or other unnecessary complexities. The tools that are used to support and test the implementation are also written in Python to allow easier maintenance and readability for any developers who are contributing to the project.

The complex *nova-schedule* code manages the virtual machine lifecycle in OpenStack. It passes requests as Python function calls to different parts of the allocation mechanism in order to make decisions. The implementation of our workload mix strategy takes the form of a *Filter* and *Weigher* module that replace comparable components in the OpenStack code. There are also a insertions or *hooks* in a small number of other OpenStack files.

When *nova-schedule* receives a request to create an instance, the nominated *Filter* code removes node choices that do not pass the current scheduling policy. The remaining nodes are then ranked by the *Weigher* to choose the node where the new instance should be loaded. The custom *Filter* and *Weigher* code allows this choice to be intercepted and manipulated to comply with the workload mix strategy.

There five main modules of the *Allocator* architecture, four of which are outlined in the architecture diagram in Figure 5.3. The fifth is the testing module.

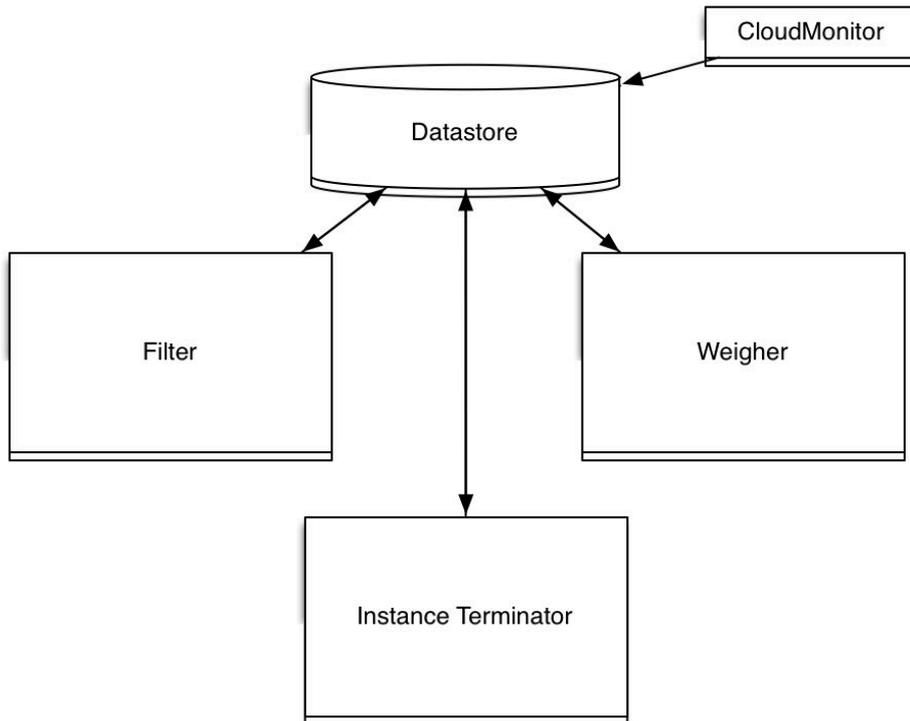


Figure 5.3: Custom Scheduler system architecture

- The *Filter* code is executed for every possible host that the new instance being created could be allocated to. The details of each host are passed, along with any *Filter* properties that the code should use as context, for example, any *scheduler_hints* that the user has specified, the code is executed and a decision is made if this node choice passes the *Filter*. The choice and the host suitability evaluated are then stored in the system data store.
- Those host choices that have passed the *Filter* stage of the *Allocator* strategy are then presented in turn to the *Weigher*, for an evaluation of their suitability. Instead of being a binary choice like the previous stage, each host is given a positive integer ranking. The node with the highest or lowest score, depending upon the current configuration of the implementation, will then be chosen to allocate the new instance. As with the *Filter* code, each choice and their resulting evaluated score are stored in the system data store.

- The terminator code is executed when a user or the system attempts to destroy a currently running instance. This code is inserted as a *hook* into the *nova-schedule* “api.py” class. In this class, code is executed to remove the currently running VM instance from the system. The *hook* will be executed when this state occurs and its aim is to record the termination and update the *Allocator*’s view of the system. It is important that the implementation maintains a current view to ensure that choices, which are inherently context sensitive, are not made with out of date information.
- The data store may be any suitable relational database, and is required to record the actions of the *Allocator* strategy, from updating choices that are presented it, to evaluations taken in order to maintain an accurate and current view of the system state.

While not featured in architectural figure above, a key part of the system development is the testing framework that developed for this thesis. This connects to the main OpenStack system and executes VM instance commands such as requesting new instances and terminating old ones. The commands are monitored as they progress through the *Allocator* software, with each *Filter* and *Weigher* decision recorded. The results of these decisions are evaluated to determine if the software is functioning as intended.

5.3 Rules language

The *Allocator* implementation relies upon an English-based rules language that defines the workload mixes that should be executed on the private cloud system. These rules are designed to be easy to read and simple to write so that system administrators may customize the performance of their deployments. The rules are used to specify actions that the *Allocator* should take such as prioritising the co-location of a particular VM type or ensuring that an incompatible set of VMs are not scheduled together.

The rules are designed to make describing workload mixes easy. Mixes normally take the style of ‘if some current situation is true **then do** some

action'. The rules attempt to capture this by using an evaluation set to decide which action should be taken. The rule defines how a host qualifies to be added to the evaluation set and then the length of the set is measured to decide which of the written actions should be executed.

A typical example rule would be:

```
FILTER:Job=CPU:<CPU,EMPTY>:1:PASS<SET>:PASS<ALL>
```

Below is an annotated version of this rule.

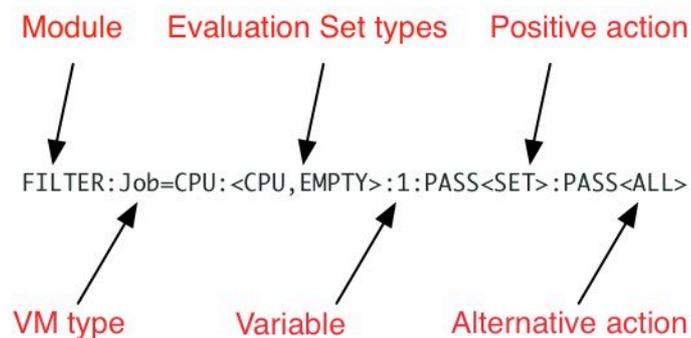


Figure 5.4: Annotated example of rule language

- **Module:** The first part of the rule defines whether this rule is applicable to either the `FILTER` or `WEIGHER` module.
- **VM type:** This defines the type of virtual machine that this rule should be evaluated for. When the user launches a VM using on an OpenStack deployment using the *Allocator* strategy, they must tag their VM with a type from one of CPU, MEM, DISK or NET. That type is passed to both the *Filter* and *Weigher* modules as a parameter and is matched against this section of the rule to see which rules should be activated and then evaluated for this VM request.
- **Evaluation Set types:** This part of the rule defines the VMs that should be added to the evaluation set. Hosts that have VMs of these types are added to the set. It is possible to define no VM types in this section, in which case all hosts would be added to the set, or any or all of up to five VM types, the four resource types and "EMPTY". In this case, hosts that have CPU type VMs or those that have non-VMs at all are added to the evaluation set.

- Variable: This number, called the variable, defines the trigger size at which the first action should be completed. If the evaluation set (defined in the previous section) size is smaller than this number the alternative action is executed. Otherwise the positive action is executed.
- Positive action: This section is called the “positive action” and its command should be executed if the evaluation set size is equal to or greater than the rule variable. In this case, it defines that all hosts within the evaluation set should be based. Other valid commands here would be:
 - PASS<ALL>, All hosts have passed and no hosts should be caught by the *Filter*.
 - FAIL<SET>, fail all hosts in the evaluation set, allowing all other hosts to pass the *Filter*.
 - FAIL<ALL>, fail all hosts, allowing no hosts to pass the *Filter*.
- Alternative action: Finally, the rule ends with the alternative action that should be executed if the evaluation set is smaller than the variable. The possible commands here are the same as for the previous section.

It is possible to have many rules in place at once, all within the *mix.rules* file placed in the *nova/scheduler/Filters/* folder of the OpenStack installation. Each rule is placed on a new line and colons split up the parameters within the rule.

Rules in this file are executed in a sequential model. If the evaluation of one rule results in a subset of the original set of hosts being passed, then the next rule is executed only on that subset.

With this simple but powerful rules language, system administrators can tailor the workload mix behaviour to their IT policies and deploy new workload mix rules as they are discovered. These rules allow the administrator to specify for example that VMs of the same type (i.e. DISK) are kept apart, if application performance is more important than the potential energy savings for co-locating them, or place them all together if a stacking strategy is preferred.

5.4 Filter

The *nova FilterScheduler* designates a *Filter* module to take the first stage of making a placement decision. Which *Filter* to use is detailed in the *nova.conf* configuration file that is normally in the */etc/nova/* folder of the main controller for the OpenStack deployment. For the *Allocator* strategy the appropriate line in the configuration file set as follows:

```
--scheduler_default_filters=Allocator
```

A detailed guide of the different options possible and the other requirements to loading custom code are included in the System Administrators Guide, provided in Appendix B of this thesis.

The *Filter* begins by querying the database for a current list of the live instances in the private cloud system. The new instance request is compared against this list to rule out any nodes that might be unsuitable. This section details how that is achieved and focuses on the *Filter* algorithm that has been devised.

In general the *Filter* attempts to remove hosts from contention where application performance will be impacted. The *Weigher* code then evaluates the remaining hosts to choose which one would be best according to the workload mix rules.

A workflow diagram of the *Filter* is presented below:

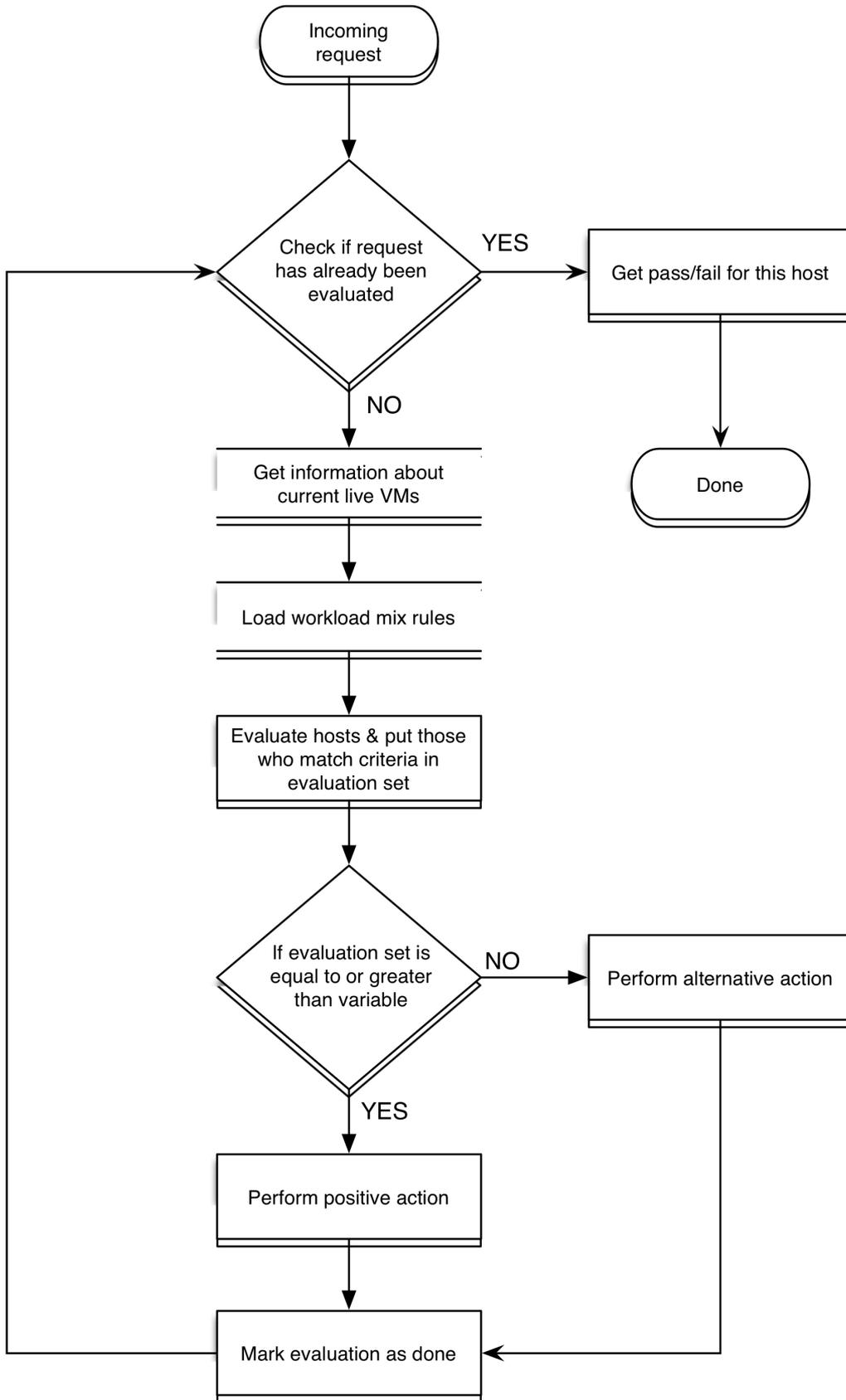


Figure 5.5: Workflow diagram for *Filter* module

To illustrate the workings of the *Filter*, and subsequently the *Weigher* in the next section, an example of the code in operation on a simplified deployment will now be discussed.

The example layout has two compute nodes: Node 1 and Node 2, with identical hardware and software. There is also a cloud controller machine, where the user will send requests for virtual machine instances. The cloud controller is running the custom *Filter* and *Weigher* code.

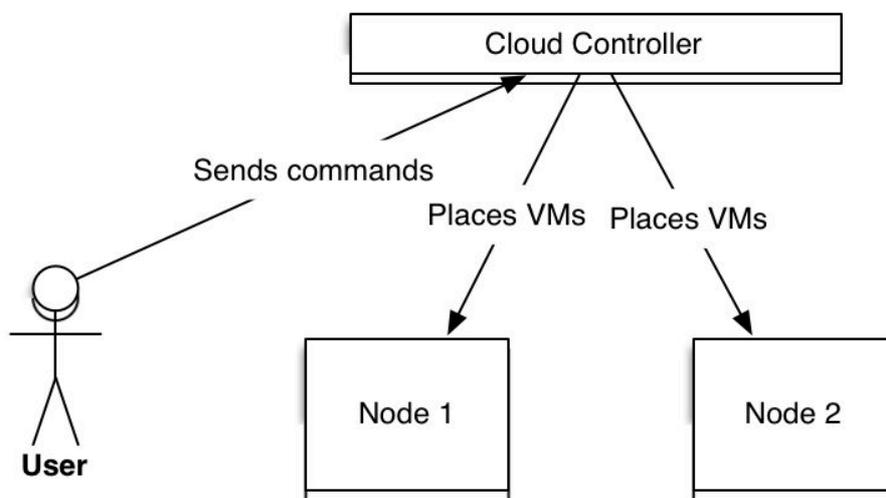


Figure 5.6: Example OpenStack deployment

The current layout on the virtual machine nodes is as follows:

- Node 1 has 3 CPU bound virtual machines
- Node 2 has no currently running VMs.

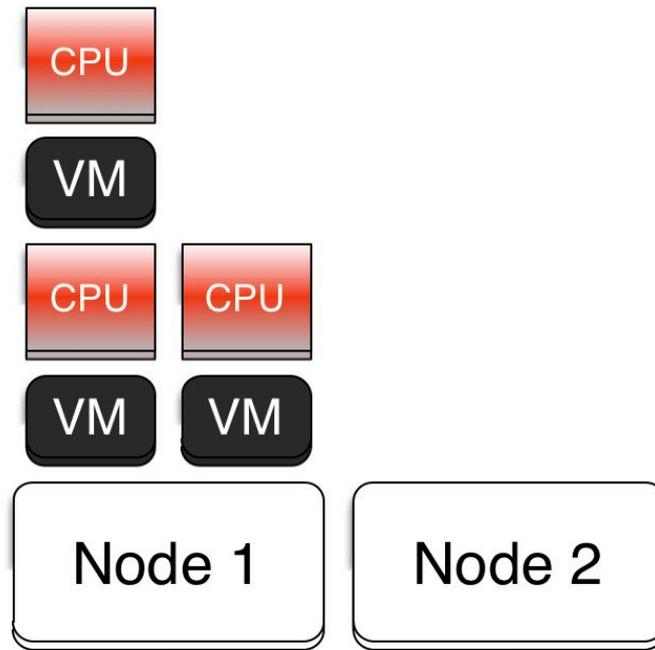


Figure 5.7: Example OpenStack Deployment current VM load

```
def allocator_filter(self, host_state, filter_properties):
    # Examine the scheduler_hints
    scheduler_hints = filter_properties.get('scheduler_hints') or {}
    vm_type = scheduler_hints.get('vm_type', [])
    # Get the request uuid and the IP address of the host
    uuid = filter_properties['request_spec']['instance_properties']['uuid']
```

Snippet 5.1: *Filter* module is invoked

When the request for a new virtual machine instance is received by the cloud controller it begins the allocation process by invoking the configured *Filter* module. The custom *Filter* module is invoked with the parameters *filter_properties* and *host_state* as seen in Snippet 5.1 above. Both parameters are dictionaries that contain context about the current state of the OpenStack system.

- *host_state*: this dictionary contains information about the host to be evaluated. For example, its IP address, capabilities, etc.

- *filter_properties*: another key/value dictionary containing any information that might be pertinent to making the placement decision. In the code above, the *scheduler_hints* are extracted to get the *vm_type* as set by the user who requested the new VM instance. Each new request has a unique identifier and this is also extracted for use in tracking the *Allocator* strategy evaluations and decisions.

The request ID is checked against the data store *FILTER_EVALS* table to see if all hosts have already been evaluated for this request ID. If not, the code will then evaluate all hosts. All hosts are evaluated at once on the first running of the *Filter* for each request. Once the hosts are all evaluated their results are stored in the implementation data store and referenced when the *Filter* is called for each host in turn. This avoids duplication of the computation.

To begin the evaluation, the *Filter* must first load the workload mix rules that it will apply to each host. The workload mix rules are loaded from a static configuration file *mix.rules* located in the *nova/scheduler/Filters/* folder of the OpenStack installation. These rules are loaded each time the custom *Filter* code is run, allowing the rules to be changed and therefore the behaviour of the entire private cloud without any re-starts of the system.

An example of a typical rule that would be valid in this situation, and is used in the prototype *Allocator* software, would be:

```
FILTER:Job=CPU:<CPU,EMPTY>:1:PASS<SET>:PASS<ALL>
```

This rule is to be used when a VM of type CPU is launched. It requires that any hosts containing VMs of type CPU or any empty hosts be added to the evaluation set. If the size of the evaluation set is greater than or equal to 1 then all hosts in that set pass the *Filter*, otherwise all hosts should pass the *Filter*.

The prototype of the *Allocator* software used in these examples and evaluated in Chapter 6 uses four rules, all variations on the example above for each job type. These rules were chosen due to the experimental results from Chapter 4 that suggested that workload tasks perform best when they are paired with similar tasks or are placed in empty hosts. Fine-grained balancing of exact placement is controlled by the *Weigher* module.

To begin evaluating this rule, the *Filter* code gets a list of the current hosts in the private cloud system and evaluating them based upon their currently running VMs, if any. Those that match the rule criteria are added to the evaluation set, a collection called *set_hosts* in the code snippet below.

```

# Get hosts that match set_rules
set_hosts = []
for rule_type in set_rule_types:

    if rule_type == "EMPTY":
        # Returns all host that have vms
        cur.execute("SELECT DISTINCT (host) FROM SCH_PLACEMENTS")
        results = cur.fetchall()

        # Put all hosts with VMs into a list
        loaded_hosts = []
        for host in results:
            loaded_hosts.append(host[0])

        # Take the list of hosts
        unloaded_hosts = hosts[:]
        # and remove those host which have VMs
        for host in loaded_hosts:
            unloaded_hosts.remove(host)
        # add theses hosts to the set hosts
        set_hosts = set_hosts + unloaded_hosts

    else:
        # For rules of a specific VM type
        # Get list of hosts with instances of this rule_type
        cur.execute("SELECT DISTINCT(host) FROM SCH_PLACEMENTS WHERE
vm_type='" + str(rule_type) + "'")
        results = cur.fetchall()
        rule_hosts = []
        for host in results:
            rule_hosts.append(host[0])
        set_hosts = set_hosts + rule_hosts
        print set_hosts

```

Snippet 5.2: Applying workload mix rules

Empty hosts are dealt with first. The list of hosts that currently have virtual machines are gathered from the data store and this list is used to create a new list of hosts that have no VMs. The workload mix rule may specify that those

hosts that have no virtual machines are added to the evaluation list, if so that is done at this point as illustrated in Snippet 5.2. Next, all hosts containing one or more VMs of the each type specified in the rule are added to the evaluation set. It is possible for the rule to specify no particular VM types to the set, in which case no hosts would be added.

Once the evaluation set is complete, its length is compared against the variable specified in the rule. If it is larger than or equal to the variable, in this case 1, the evaluation set is said to be positive and the appropriate action is executed.

The actions are defined at the end of the rule. In the example given, the actions are either to pass all hosts within the set or pass all hosts. Our current load is that Node 1 has 3 CPU based VMs and Node 2 has none. Applying the *Filter* to this situation would result in both Node 1, which has CPU based VMs, and Node 2, which is empty, being added to the evaluation set. The evaluation set is therefore size 2 in length and this is larger than the variable, which in the rule was set to 1. The first action will therefore be executed, code snippet 5.3 shows the PASS<SET> action being executed.

```
# Complete the action, whatever that might be.
if action == "PASS<SET>":
    # The most common action is to pass the hosts that met the rules
    for host in set_hosts:
        # now need to mark the host as suitable for this uuid
        cur.execute("INSERT INTO FILTER_DECISIONS VALUES ('" +
str(uuid) + "', '" + str(host) + "', 1)")
```

Snippet 5.3: Executing an action

The first action in this case is to pass all the hosts in the evaluation set. To do this, in the data store each host in the set is marked as having passed the for this instance creation request ID. Once the evaluation is complete, the request ID is used to mark that the evaluation is done in the data store as well.

The evaluation part of the *Filter* is only executed once per request ID. The final part of the code is executed for every host that is to be evaluated in turn. The host IP address and requested ID are looked up and the result which is stored

is returned as the pass or fail value for this host on the *Filter*.

The *Filter* has now evaluated each possible host in the system against the workload mix rules and eliminated those that are not desirable placement locations. For the worked example, both example hosts have passed this *Filter* for the current rules based on their running VM states.

5.5 *Weigher*

The *Weigher* code evaluates those hosts that have passed the *Filter* stage and assigns each a weighted score. It begins by gathering all the information required to make its informed decision. There are two pieces of critical data: the list of the currently running virtual machines and a description of the new instance to be placed. The former is pulled from the data store and the latter is provided to the *Weigher* by the *scheduler_hints* functionality within OpenStack. With those two pieces of data, it is possible for the *Weigher* to appropriately evaluate each potential placement location according to the workload mix rules. In the code snippet 5.4 we can see the *Weigher* gathering data from these parameters.

The workflow diagram for the *Weigher* module is presented in Figure 5.8:

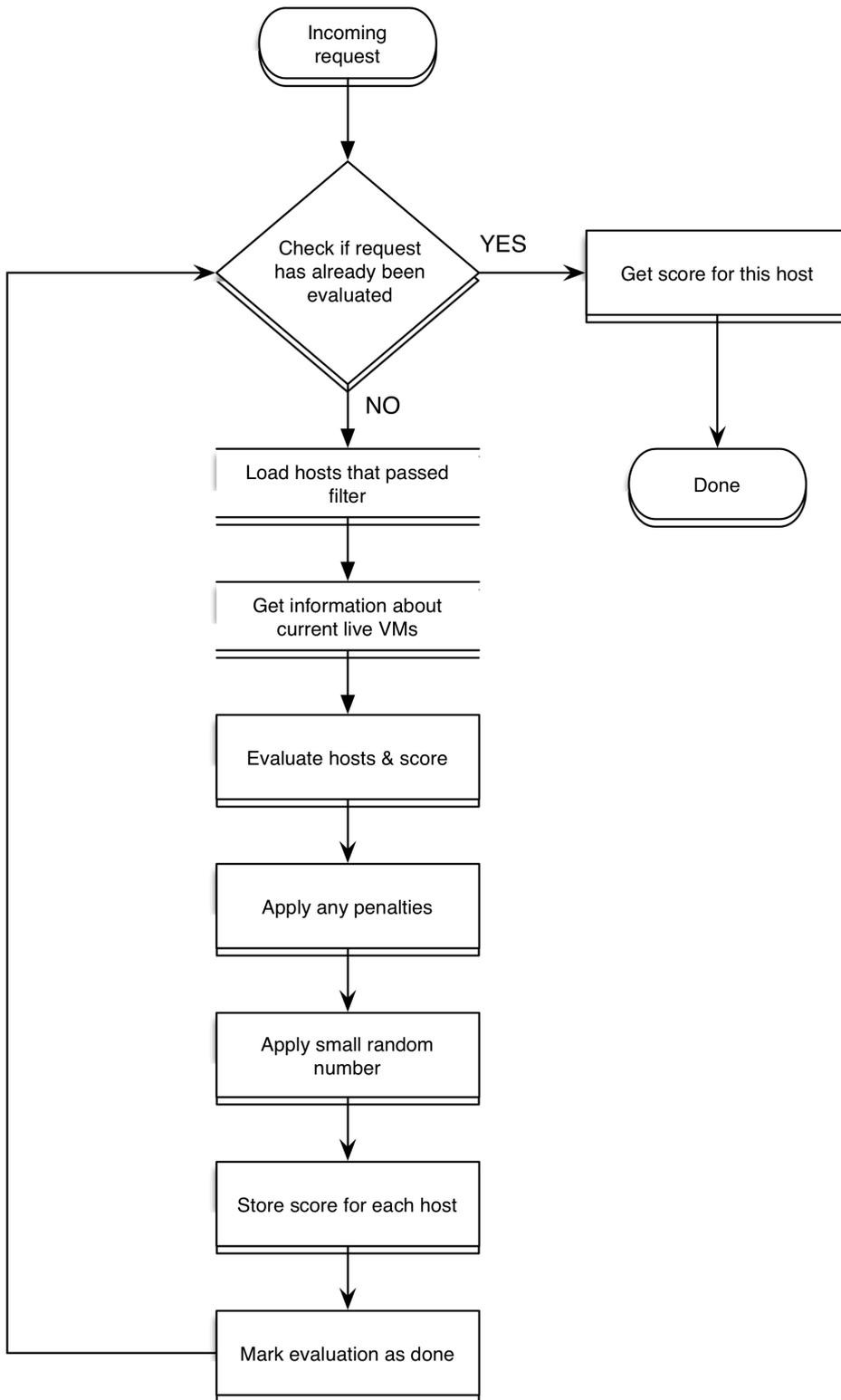


Figure 5.8: Workflow diagram for *Weigher* module

```

def compute_allocator_cost(host_state, weighing_properties):
    # Examine the scheduler_hints
    scheduler_hints = weighing_properties.get('scheduler_hints')
    vm_type = scheduler_hints.get('vm_type', [])

    # Insert weigh request into WEIGH table
    request_id =
    weighing_properties['request_spec']['instance_properties']['uuid']

```

Snippet 5.4: Weigher module is invoked

The *Weigher* code is called by *nova-schedule* for each host that has passed the previous *Filter* stage. The code is passed two parameters, *host_state* and *weighing properties*.

- *host_state*: as with the *Filter* module, the *host-state* variable includes information on the host to be evaluated.
- *weighing_properties*: the *weighing_properties* dictionary is similar to the *filter_properties* dictionary passed to the *Filter* module. It contains context that may be relevant to the weighing decision.

From these variables, the *scheduler_hints* passed *vm_type* is extracted along with the request ID and the host IP address. The request ID is then checked against the data store table *WEIGH_EVALS* to see if host weighing has already occurred for this request ID. If so the evaluation stage is skipped to avoid duplicate computation.

The worked example in this chapter has Node 1 containing 3 CPU based virtual machines and Node 2 being empty when a new request is issued to create an additional CPU based VM.

The *Weigher* module is called by *nova_schedule* for each of the hosts that passes the *Filter* stage. However, as the computation takes place on the first time the *Weigher* module is called for each request ID, the *Weigher* code must retrieve the complete list of passed hosts from the custom implementation data store.

Those hosts are then compared with the list of hosts with currently running VM instances to create separate lists of loaded and unloaded hosts that have passed the *Filter*, as seen in snippet 5.5. Hosts with currently running VMs are collected from the data store and the returned list is compared with the list of hosts that passed the *Filter* module for this request to identify those hosts that are unloaded.

```
# Get list current hosts with VMs
cur.execute("SELECT DISTINCT(host) FROM SCH_PLACEMENTS")
results = cur.fetchall()
loaded_hosts = []
for host in results:
    loaded_hosts.append(host[0])

# and list of vms without hosts
unloaded_hosts = passed_filter_hosts[:]
for loaded_host in loaded_hosts:
    if loaded_host in unloaded_hosts:
        unloaded_hosts.remove(loaded_host)
```

Snippet 5.5: Get list of current hosts with VMs and use list to identify unloaded hosts

Each host is now evaluated and assigned a weighted score. To begin, unloaded hosts are assigned a value of 16. In the worked example, Node 2 would be given a score of 18 at this point. The scores for this evaluation are in table 5.1 and the code to calculate this evaluation is highlighted in snippet 5.6.

Node	Score
Node 1	0
Node 2	16

Table 5.1: Score of example after unloaded hosts are evaluated

```

# Setup scores for each host
for host in passed_filter_hosts:
    score_dict[host] = 0
    # Evaluate a score for each host
    # If host is empty, score = 16
    if host in unloaded_hosts:
        score_dict[host] = 16

```

Snippet 5.6: Assign unloaded hosts the score 16.

Then, if a host has one instance of the same *vm_type* as the instance that is about to be launched, it is assigned a score of 20. In the worked example, Node 1 has a CPU based virtual machine, so it is assigned a score of 20, as recorded in table 5.2 and computed in code snippet 5.7.

Node	Score
Node 1	20
Node 2	16

Table 5.2: Score of example same instance types are evaluated

```

# If host has one instance of same vm_type, score = 20
cur.execute("SELECT count(*) FROM SCH_PLACEMENTS WHERE host = '"
+ str(host) + "' AND vm_type = '" + str(vm_type) + "'")
count = cur.fetchone()
count = int(count[0])
if count >= 1:
    score_dict[host] = 20

```

Snippet 5.7: Assign hosts with one VM of the same type the score 20.

Then, if the host already has more than 1 VM on the same type, the score is reduced by two for each additional virtual machine. In the worked example, Node 1 has a total of 3 CPU type virtual machines, so its score would now be 16. The code to apply this penalty is illustrated in code snippet 5.8.

Node	Score
Node 1	16
Node 2	16

Table 5.3: Score of example after hosts with multiple VMs of the same type are evaluated

```
# Remove 2 from score for each additional vm of same type
if count > 1:
    #raise Exception ("count = " + str(count))
    penalty = count - 1
    penalty = penalty * 2
    score_dict[host] = score_dict[host] - penalty
```

Snippet 5.8: Remove 2 from score for each additional VM of the same type

Next, a score of five is removed for any VMs of a different type on that host. In the worked example there are no VMs of different types to the one that is to be created on any hosts, so the scores are unchanged, the code to apply this additional penalty is provided in snippet 5.9. The penalty values shown here are those that have been concluded after experimentation during the development of this software. Different values for each penalty were evaluated to determine the final implementation. The evaluation of this software (Chapter 6) is based on code with the values shown here.

```

# Remove 5 for different type vms on each host
cur.execute("SELECT count(*) FROM SCH_PLACEMENTS WHERE host = '" +
str(host) + "'")
total_count = cur.fetchone()
total_count = int(total_count[0])
if total_count > count:
    penalty = total_count - count
    penalty = penalty * 5
    score_dict[host] = score_dict[host] - penalty

```

Snippet 5.9: Remove 5 from score for each additional VM of a different type

The scores that are assigned by the *Weigher* are based on lessons learned by the experiments in the workload mix chapter (Chapter. 4). The conclusions found that co-located virtual machines across all four types gave the best energy scores.

Performance for co-located machines was also improved for CPU, MEM and NET types. DISK VMs performed at approximately 100MB/s when co-located, but dropped to less than 50MB/s when paired with a CPU intensive VM. To avoid this, the preferred strategy for this *Allocator* strategy is to bias the allocation towards co-locating virtual machines of the same type.

Finally, the implementation is faced with a situation where two hosts have the same score and therefore either one is a suitable choice for the VM placement based on our workload mix rules. Due to way that the *Weigher* module is called by the *nova-schedule* code, it is not possible to predict which host will be chosen by OpenStack if both hosts are returned with the same score. As our *Allocator* relies upon knowledge of the system, including accurate positions of VM placements, this would lead to uncertainty and degraded performance of the allocation strategy.

To fix this problem, a small random variable is added to the final *Weigher* score as can be seen in snippet 5.10.

```
# Add a small random value to each score to avoid conflicts
random_var = random.uniform(0, 0.1)
score_dict[host] = score_dict[host] + random_var
```

Snippet 5.2: Add random number to score to avoid conflicts

This value, between 0 and 0.1 is added to each score. It allows the implementation to definitely have an answer as to which host should be chosen by OpenStack and allows the choice to be recorded in the data store.

To recap, the pseudo code for the evaluation of each host is:

```
If host is empty
    assign is a score of 16
else
    if host has one instance of same VM type as we are attempting to
    schedule
        score should be 20
    remove 2 points for each additional vm of the same type
    remove 5 points for each additional vm of a different type
```

Snippet 5.11: Pseudo code for *Weigher* module evaluation

As before with the *Filter* code, all evaluations for each host are recorded in the database, and when complete the request ID is marked as evaluated so that the computation is only done once.

When the *Weigher* module is called for each host in the system, the host is looked up in the data score and its weighed score is returned for each host. If that score is the best score yet seen for this request ID, the data store is updated with the choice.

Once all hosts have been returned with a score to the *nova-schedule* module, the virtual machine is placed on the host with the best score. The data store contains a representation of where each virtual machine is placed so that it may accurately evaluate each new request.

The values used in this implementation of an *Allocator Weigher* module are unique to the workload mix that is being evaluated. The numbers were chosen based on the experimental results of workload mixes on this hardware and software configuration that is detailed in Chapter 4. We found that the optimal workload mix for these workloads on these hosts could be found by generally allocating no more than 3 VMs of the same type to a single host. Through a process of experimentation, the values used in this example were found to give the best implementation of this strategy.

For the purposes of the prototype software the values in this example are hard-coded into our *Weigher* module. For a more generalised, production ready implementation of *Allocator* these values would be read from a configuration file and be adaptable based on the workload mixes that should be applied.

5.6 Data store and decision tracking

Data storage is key part of the *Allocator* architecture. A representation of the current of state of the private cloud system is required at all times for the allocation strategy to remain effective. Any inaccuracies could lead to reduced system performance, application lock-ups or starvation and increased rather than reduced energy consumption.

The data store contains information from each key stage of the scheduling process, maintaining records of *Filter* requests, evaluation of hosts and final decisions as well as the same data for the *Weigher* module. This is in addition to the model of the current system status.

By maintaining a record of each decision taken it is possible to view the implementation execution in real time as users request and then terminate virtual machines. Without the data store, such information could only be obtained by trawling the OpenStack *nova-schedule* log, which contains a vast amount of information, most of which is superfluous to the task of scheduling.

The design of the *Allocator* implementation is such that any data storage

facility would be suitable for the implementation; for the purposes of this prototype development a relational MySQL database instance was used. MySQL was used in the example in this chapter as the OpenStack architecture upon which this implementation is built uses MySQL to track the state of all VMs in its system. We add only a small amount of additional load onto this database (as can be seen by the lightweight amount of data store in the tables below). As such the actual *nova-database* that is used by OpenStack may also be used for the *Allocator* implementation. As well as avoiding the introduction of additional data store into the computing environment it this decision means that the *Allocator* has the same expansion limitations that already exist for OpenStack. If the limits of the database are being reached in terms of concurrent connections, traffic or data size then a new store will be required for the entire system, not just *Allocator*.

In this relational database model, the data store maintains 8 tables for the implementation:

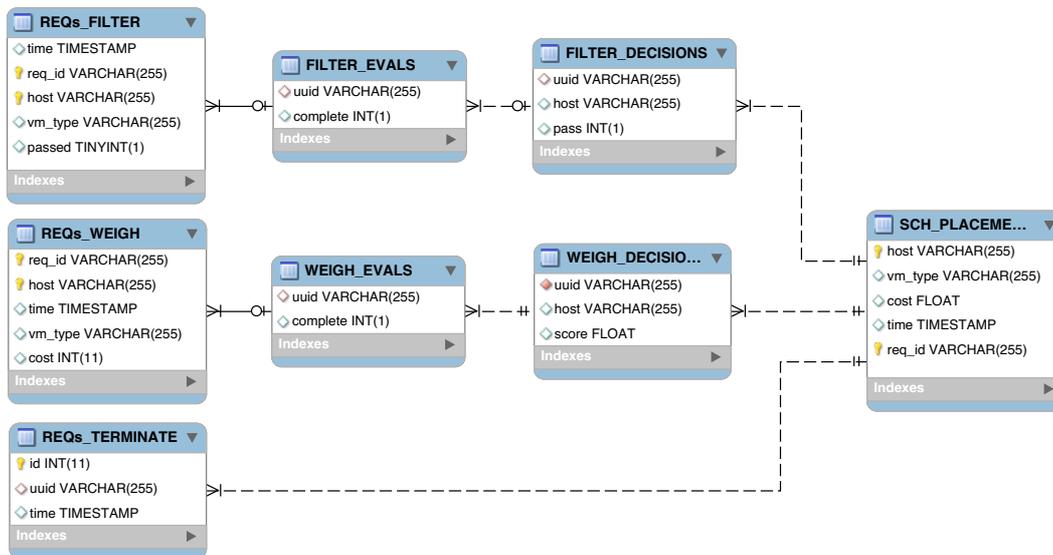


Figure 5.9: Schema of implementation data store

- The *SCH_PLACEMENTS* table represents the implementation's master view of all the currently running virtual machines in the private cloud system. It is updated by the *Weigher* to record the compute node that has been evaluated with the *best* score, as this is the one that will be chosen by the OpenStack *nova-schedule* module. VM termination is also tracked through the various API calls and when a termination command is

issued the table is updated to remove whatever VM has been destroyed.

- *REQs_FILTER*: This table stores all requests by *nova-schedule* that are passed to the *Filter* module. These contain information about the host, represented here by its IP address, and the request's unique identifier.
- *FILTER_EVALS*: The evaluations for all request processed by the *Filter* are stored here, as they are evaluated on the first time the *Filter* code is executed for this request ID.
- *FILTER_DECISIONS*: When OpenStack *nova-schedule* calls the *Filter* for each host for each request ID, this table stores the value that is returned to allow tracing if there any discrepancies between this and the values that were calculated on the evaluation processing when this request ID was first encountered.
- *REQs_WEIGH*: As with *REQs_FILTER*, all requests received by the *Weigher* module from *nova-schedule* are stored here to all for logging. The table contains information about each request ID and the host to be evaluated.
- *WEIGH_DECISIONS*: Again, all costs that are evaluated by the *Weigher* module are stored here after they have been processed the first time the module receives this request ID.
- *WEIGH_EVALS*: This table stores the weighed values that are returned to OpenStack for each host.
- *REQs_TERMINATE*: Finally, all requests by the system or user to terminate VM instances are stored here to allow tracking on any errors or bugs.

These 8 tables provide enough information to track the allocation strategy decisions from the users first request through each possible placement location being evaluated to the final placement decision.

An example of the data that is collected during each decision is shown in Table 4, below.

Request ID	006b390c-b3fc-4e4c-834b-c4190404beca
Host being evaluated	cloud-node-33
Timestamp	2013-08-27 10:51:56
VM Type	DISK
Score	16

Table 5.4: Example of REQs_WEIGH entry

Table 5.4 displays an entry for host that has been weighed by the custom *Weigher* module. Here, the request unique identifier, host identifier, timestamp and *vm_type* are stored along with the evaluated score for these conditions relative to the current state of the private cloud.

5.7 Tracking termination of VMs

It is necessary to track the termination of virtual machine instances to maintain a current and accurate view of the system status. Without such a view of where VM instances are placed and their current types, the *Allocator* strategy would be blind when attempting place new VM instances. In such a situation, VMs could be placed where there application performance or energy usage is severely impaired because of an unknown instance or instances also placed on that host.

To track what VMs are terminated, a *hook* of custom code is inserted into the *nova-schedule* `api.py` class. All modules within *nova*, including the front-end user facing interfaces, communicate with other modules through the API. Therefore, it was deemed that this is the best place to intercept and run custom code when the user or system requests that a VM termination command is issued.

```

def _delete(self, context, instance):
    try:
        # Insert into termination request into REQS_TERMINATE table
        cur.execute("INSERT INTO REQS_TERMINATE(time, uuid) VALUES('" +
str(datetime.now()) + "', '" + str(instance_id) + "')")

        # Remove this uuid record from SCH_PLACEMENTS
        cur.execute("DELETE FROM SCH_PLACEMENTS WHERE req_id = '" +
str(instance_id) + "'")

```

Snippet 5.12: Track deletion of VMs and update the data store when they occur

Once placed in the correct location within the *nova* codebase, the tracking code is quite simple. In the `_delete()` method, code has been added to update the data store with a record of the termination request in *REQs_TERMINATE* and remove the VM record from *SCH_PLACEMENTS*, this is illustrated in snippet 5.18.

This is sufficient to track what requests have been made, for the purposes of logging and debugging while maintaining the *Allocator's* master view of the private cloud system.

5.8 Deployment and use

The *Allocator* of workload mix allocation strategy was deployed with the fifth release of OpenStack (codenamed “*Essex*”) on dedicated servers in the University of St Andrews Computer Science department server room.

The servers, part of the St Andrews cloud computing co-laboratory (StACC) are 2010 model Dell Powerededge R610 Servers with 2 Intel Xeon E5620 Processors, 16GB of RAM and a Seagate Savvio 10K 6-Gb/s 146GB Hard Drive.

To deploy the *Allocator*, our system administrator installed a standard version of OpenStack from a pre-made image. The same image was applied to each compute node, which would not be modified and a different image for the

machine designated “*cloud-controller*” that would be modified by the *Allocator* code.

The code is customized by pulling the latest version of the *Allocator* from Github.com³⁰ and over-writing some parts of the OpenStack system. The *git* source control system is used to automate this process, but it can also be completed manually by downloading the code and replacing the appropriate files.

A system restart is then required, and as OpenStack has many services which all must be re-started in the correct order, a script was written that allowed that to be completed easily³¹.

Once the system was deployed it ran over a three-month period while being evaluated and was available to StACC users. The *CloudMonitor* software was deployed alongside OpenStack to provide cross correlation of its results. *CloudMonitor* data was stored alongside the *Allocator* decision tracking in our data store, which was backed up daily and stored offsite. Additionally the database was manually backed up before any major upgrades or experiments.

This chapter has shown the design and development the *Allocator*, an example of a workload mix allocation strategy for OpenStack. The effects of the implementation on virtual machine performance and energy usage are covered in the next chapter.

³⁰ <http://github.com/jws7/allocator>

³¹ <http://github.com/jws7/allocator/nova-restart.sh>

6 Evaluation of OpenStack *Allocator*

6.1 Introduction

This chapter discusses the evaluation of our workload allocation software, *Allocator*, when compared to the standard VM allocation strategies for OpenStack. The aim of our work was to implement the workload mix lessons from Chapter 4 by developing an *Allocator* for OpenStack that reduces energy consumption without significantly affecting application performance.

The experiments in this chapter follow a similar pattern to those in Chapter 4 where the *Phoronix* benchmarking suite is used to measure application performance from within virtual machines and *CloudMonitor's* energy monitoring capability pulls live energy usage metrics from PDUs connected to the host hardware.

For the experiments described in this chapter all virtual machines were launched and assigned by the *Allocator* software from Chapter 5, or the alternative OpenStack allocation strategies that are described in Section 6.3.

The experiments in this Chapter were conducted by the *ExperimentRunner* software described in Section 6.2. This software represents another significant engineering effort for this thesis and provides a tailored approach to managing virtual machine creation, deletion and instrumenting the experiment process.

The data collected by *ExperimentRunner* and *CloudMonitor* are presented and analysed in Section 6.4. All analysis was conducted in the statistical package R. A discussion of the results of these experiments concludes the chapter in Section 6.5.

6.2 Experimental design

Our experiments aimed to evaluate the effectiveness of *Allocator* software for OpenStack. The software, which is described in Chapter 5, attempts to reduce the energy consumption of a private cloud system while maintaining application performance. This is in contrast to traditional approaches to green computing that consolidate workload to reduce the number of physical machines required and therefore reduce the total amount of energy consumed. These approaches frequently do not take into consideration the impact that consolidation will have on the performance or throughput of that workload that is being executed. Such approaches may also impact dependability if VMs are consolidated onto a low number of VMs, as the loss of a physical host will result in the failure of a larger percentage of VMs.

The experiments discussed in this chapter use synthetic benchmarks to represent real world tasks being executed on a private cloud system. Benchmarks are used because they provide a clear performance metric that can differentiate one computing environment when compared with another. This differentiation is important when comparing the performance of identical virtual machines that have been configured and arranged in different ways.

Throughout the experiments in this chapter, the hardware, software and type of virtual machines are all constant. The only variable is the allocation strategy that is used by OpenStack to arrange the virtual machines. For each allocation strategy a series of tests are executed with combinations of workloads.

The experiments were conducted for 25, 50 and then 75 percent system utilization. At 0% no work is being conducted, and at 100% utilization the variation in system performance would be negligible, as all hosts would be maximally utilized. There may be some variation in performance depending on the VMs that are co-located at 100%, but this is left as future work.

The rest of this section now discusses the hardware and software configurations for these experiments, including a discussion of the *ExperimentRunner* software and modifications to Ubuntu and OpenStack that were necessary to achieve optimal performance from our test platform.

6.2.1 Hardware setup

The experiments discussed in this chapter were running on four hardware nodes; 2010 model Dell *Poweredge R610* Servers with 2 Intel Xeon E5620 Processors, 16GB of RAM and a *Savvio 10K 6-Gb/s 146GB* Hard Drive, running Ubuntu 12.04 OS. Power measurements were taken for each of the devices using the same hardware PDU discussed in Chapter 4 and the data was collected in the same way using *CloudMonitor's EnergyMonitor* module.

Section 6.2.3 discusses the configuration of the software systems used on this hardware, including Ubuntu 12.04 and OpenStack version Essex.

6.2.2 ExperimentRunner software

To conduct the experiments for this chapter a software tool was developed that allowed the distributed coordination of each benchmark on multiple virtual machines. The *ExperimentRunner* software described in this section was used to conduct each stage of the experiment, from management of the VM lifecycle, to running benchmarks and finally termination of VMs. Each step in the experimental process was instrumented and recorded in a MySQL database. By the completion of this work over 24,000 runs of the experiments had been conducted and 22 million power measurements had been taken over 9 months.

The *ExperimentRunner* software has evolved over the course of this work, expanding to include modules to evaluate different configurations of workload allocations. It began simply, utilizing a software library to execute and parse command line Linux programs on remote systems.

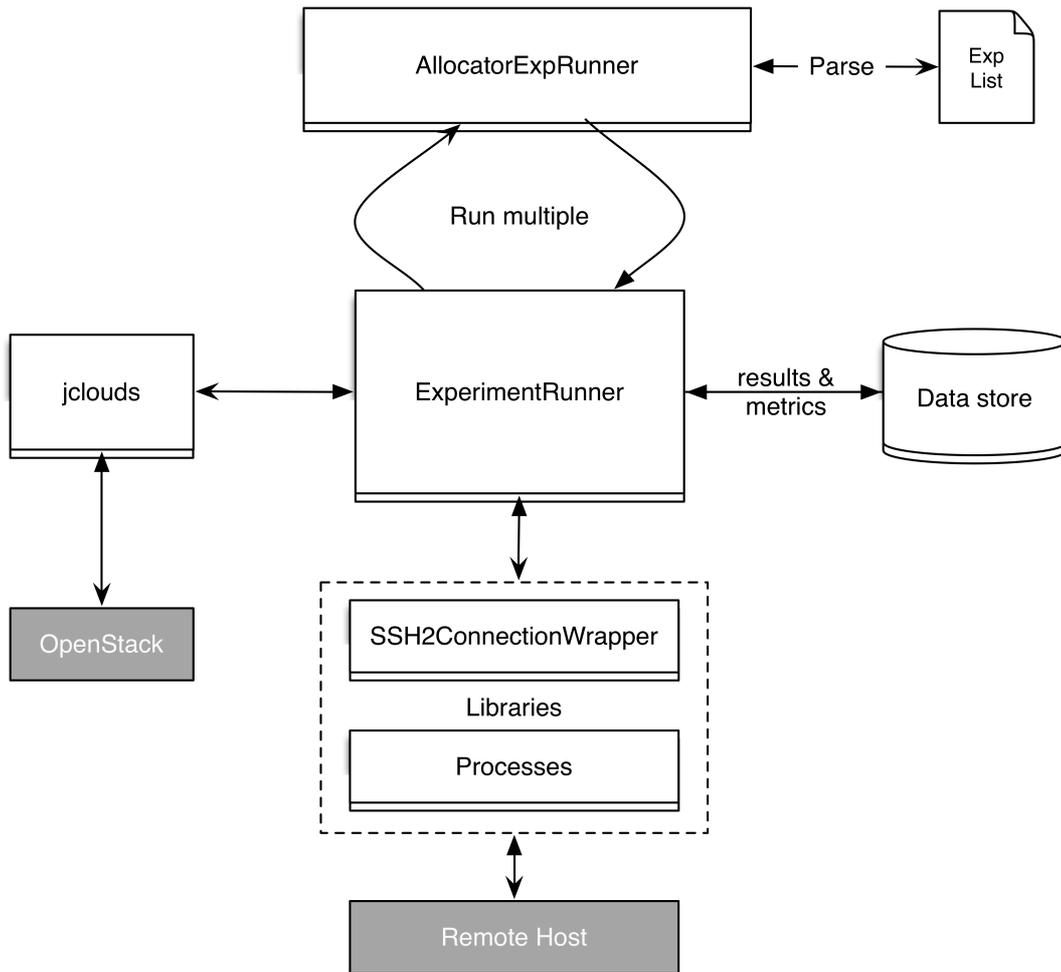


Figure 6.1: Architecture diagram of the *ExperimentRunner* software

The software libraries that became the basis of *ExperimentRunner* were *SSH2ConnectionWrapper* and *Processes* developed at St Andrews by Graham Kirby. The *SSH2ConnectionWrapper* creates a connection to a desired host using specified credentials and the connection is then used by *Processes* to remotely execute a command. On a command line interface, executing a command locally may produce some textual output, which is captured in a *ByteArrayOutputStream* when executed remotely by the *Processes* module. The code for doing this can be seen in Snippet 6.1.

```
// Open an SSH Connection to the remote server
this.ssh = new SSH2ConnectionWrapper(host, sshUser, sshPwd);
// Create output stream to use
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
// Execute remote process through SSH connection
Process runProcess =
Processes.runProcess(command, this.ssh,outputStream, System.err);
```

Snippet 6.1: Connecting to a remote server and executing a command

The *ExperimentRunner* software uses this functionality to remotely log in to an OpenStack VM, execute a *Phoronix* benchmark command and capture the output. The text returned by the byte array is parsed to gather the benchmark results and stored in a database along with the current timestamp and details of the machine it was run on.

Once the basic functionality of executing remote commands was in place, *ExperimentRunner* was extended to execute a command continually for a specified time-period. The ability to execute benchmarks constantly for a specific period was vital to achieve the intended experiment of running different tasks together to evaluate their characteristics when co-located. The development of the functionality enabled the Workload Mix experiments described in Chapter 4 to be run efficiently and be reproducible. The aims of those experiments also influenced the ability of *ExperimentRunner* to run multiple threads of execution at once to allow different benchmarks to be executed simultaneously on different remote machines.

An example of the code to run experiments for a specific period is shown in code Snippet 6.2:

```
// Run while time has not expired
while(System.currentTimeMillis() < (startTime + timeToRun)){
    // Create runner obj with params (exp, debug, host)
    ExperimentRunner runner = new
        ExperimentRunner(new Experiment(cmd), true, host);
    runner.runExperiment();
}
```

Snippet 6.2: Running an experiment for a specified length of time

Providing virtual machine lifecycle management was the next important addition to the *ExperimentRunner* software stack. In order to properly test and evaluate the *Allocator* VM placement strategy for OpenStack, *ExperimentRunner* had to be able to interact with OpenStack to launch virtual machines, track their progress and terminate instances at the end of each experiment run. *ExperimentRunner* must translate an experiment plan into a series of VM launches and terminations and within those VMs run the *Phoronix* benchmarks that evaluates the workload placement strategy.

The ideal solution to provide this functionality would be a Java library for the OpenStack API. Such a library, *jclouds*, exists but is feature incomplete in terms of the functionality required to conduct the experiments particular to our *Allocator* workload allocation strategy.

jclouds is an open source library for the Java language that provides interactions with the REST-like APIs employed by 30 cloud providers, including Amazon, Azure, Rackspace, and OpenStack. It provides easy to use methods for launching and managing virtual machines. Unfortunately the API is quite limited in its use of some of the more obscure OpenStack features, such as the *scheduler_hints* functionality that our workload allocation strategy relies upon.

This limitation prevents the use of *jclouds* as a full solution to the VM management needs of *ExperimentRunner*, so discussions were held with one of the key developers of the *jclouds* library in early 2013. They reported that while support of *scheduler_hints* was part of the *jclouds* roadmap, it would not be in their immediately forthcoming releases. Therefore an alternative solution was

sought that would allow full interaction with OpenStack within the time that we had to complete this work.

All of the commands that were required to properly test and evaluate the *Allocator* software can be issued from the command line using the standard OpenStack tool *nova-tools*. As the previously developed *ExperimentRunner* module contained code to remotely execute and parse Linux commands, it was decided that an appropriate modification to the *SchedulerExpRunner* module would be to utilize this functionality to interact with OpenStack via the command line. The functionality could allow *ExperimentRunner* to issue *nova* commands to launch and manage virtual machines and parse any response that was issued by OpenStack. *jclouds* would still be used in the final edition of *ExperimentRunner*, as an easy way to monitor the current status of virtual machines and report information, such as their public IP address.

Once OpenStack receives a command from *nova-tools* to boot a new VM instance, it replies with a table of information about the new instance. One piece of that information is crucial to track the VM instance and therefore be managed by *ExperimentRunner*: its universally unique identifier.

Universally unique identifiers (UUIDs) allow individual items to be uniquely labelled by a 128-bit number made up of 32 hexadecimal digits. The standard, developed by the Open Software Foundation, specifies that these numbers are “*practically unique*” if properly randomized due to the very large number of possible combinations (3.4×10^{38} possible numbers).

To parse the UUID from the *nova* tools response it would be necessary to collect the entire response as a string and use a regex to locate the UUID within.

Adding interaction and the ability to track and delete VMs to *ExperimentRunner* was the last stage in preparing the software to conduct our experiments. The final edition of the software can launch and manage any number of virtual machines remotely and log into those VMs to deploy our benchmarking tools. All of the processes of *ExperimentRunner* and results of the tests are instrumented and stored in a remote, replicated database for analysis. Section 6.4 discusses the experiments that were executed and

presents an analysis. The results are discussed in Section 6.5.

6.2.3 Software system configurations

In order to achieve optimal performance from OpenStack VMs a number of modifications to configurations were required on both the guest and host software systems. The details of these modifications are discussed in this section. The operating system used for both the guest and host machines was Ubuntu 12.04, the OpenStack version was Essex.

6.2.3.1 *raw* vs. *qcow2* images

OpenStack defaults to using compressed disk images for its virtual machines. The advantages of compressed images are obvious: they take up less physical space on the storage device, can therefore be transmitted easily across the network, and are widely interoperable with other cloud software stacks like Amazon EC2.

OpenStack favours the *qcow2*, a compressed image format that will grow the required space when data is written (*qcow2* stands for QEMU Copy on Write). A *qcow2* image that has had 1GB of data written will be 1GB in size, regardless of the size that the partition appears to the guest VM. When more data is written, the image is expanded to accommodate the new data.

However, as the experiments discussed in this chapter are based on benchmarking tools, best performance from all of the software systems was required in order to get a true measurement of the baseline performance and how subsequent changes in configuration would affect the performance.

For this reason, the experiments in this chapter were conducted on VMs using the *raw* disk type. A *raw* image must be the size of the partition it will be allocated to, for a 10GB partition this would be 10GB in size, regardless of how much data is written to it. The benefits of this format are that it does not need to be expanded when new data writes occur, improving throughput and I/O performance. More accurate measurements of underlying performance are therefore possible.

Usefully, OpenStack allows the configuration of its system to take advantage

of the best features of both compressed and *raw* images. *Nova* allows compressed images to be stored and transmitted between hosts and for those images to be converted at boot time to *raw* images by the host *nova-compute* daemon. Configuring the system in this way allowed for the best VM performance, but also maintained the advantages of small image sizes to be transported and stored around the system.

qcow2 is a more commonly used disk format type, particularly on OpenStack, but the penalty to resize the disk as I/O operations occur dominates the performance. It is not therefore possible to gain a realistic insight into the relative disk performance of a workload mix when using a compressed format. For that reason, the experiments in this chapter use VMs with a *raw* disk type.

Prior to switching to *raw* the default OpenStack *qcow2* images achieved ~10MB/s disk write speed, after the modification that speed was ~20MB/s.

6.2.3.1 *Disk elevators*

Another modification of the virtual machines in the pursuit of greater I/O throughput was the adjustment of the disk *elevator* algorithm to an optimal configuration.

The *elevator* of a disk is the scheduling algorithm used to determine the behaviour of the physical disk components when writing and reading data. It is named after elevators in buildings that can have different configurations determining how they respond to requests from different floors to transport passengers. A different elevator algorithm will result in different hard drive seek times, a factor that can vastly impact the I/O performance and throughput.

The *CFQ* (completely fair queuing) algorithm, set as default on Ubuntu 12.04, uses the submitting processes' priority level to schedule disk writes. Each thread is assigned a time-slice within which it can submit I/O tasks, and each thread gets a fair share of the time slices.

An alternative strategy is *deadline*, a latency-orientated I/O scheduler. Here,

requests are sorted by the sector of the hard drive they affect so that they can be dealt with in turn as the head passes over sectors in the disk. Each request is given a deadline, which if it expires immediately boosts the priority of that request so that it is handled more quickly. Our initial performance tests showed greater performance for *deadline* over *CFQ* on our hardware, so the former was chosen for the host operating system.

As the guest VMs are running on top of a host system, it makes sense for them to use no scheduling algorithm at all and simply pass their requests down to the host. This is what the *noop* trivial scheduler does, and as such it was set for the guest operating systems.

Prior to setting the guest and host elevators in this way, disk performance was ~20MB/s; this increased to ~28MB/s after the modifications.

6.2.3.2 *Libvirt XML template*

The StACC installation of OpenStack uses the KVM hypervisor to provide virtualization functionality. The *nova-compute* daemon interacts with the KVM hypervisor through the *libvirt* API, a C library available on most Linux distributions with bindings to a number of popular languages.

libvirt uses an XML template to configure virtual machines. The template governs how the guest virtual machines are launched, configured and what devices are attached to them. Two important commands for optimizing performance that are controlled by the template are the disk cache and I/O modes.

There are two main modes of caching for KVM guest virtual machines: *writethrough* and *writeback*. These modes affect how data writes are reported to the guest machine. With *writethrough* the guest is only notified that data has been successfully written with the write is completed by the host storage system. This incurs delays, as the guest system has to wait for the host to confirm that the write has indeed occurred safely, and the added level of complexity traversing a hypervisor layer is slower than a traditional I/O write operation.

writeback adjusts the cache mechanism to report the write as completed as soon as the host receives the data in its page cache and perhaps before it begins the write to disk. This means that the report occurs sooner in the write process, and allows the guest VM to continue its operations without waiting for full confirmation.

Setting the cache mode to *writeback* improves guest VM performance, but is obviously a riskier mode than *writethrough* as the VM may believe that data has been successfully written when it may not have been. In a situation for example that the host system experiences a power failure, the guest VM may experience data corruption after a reboot.

To maximize performance a previous modification, detailed in Section 6.2.3.1, we set the disk images used for the guest VMs to the *raw* type. When *raw* disks are used better I/O performance can be achieved by setting the *libvirt* driver attribute “*io*”, which governs asynchronous I/O policy, to *native*. Setting the policy to *native* rather than the alternative *threads* can lead to data corruption in compressed disk formats like *qcow2* due to bugs in older versions of the Linux kernel, for this reason it is disabled by default on most versions of *libvirt*. As we are using *raw* types, we can safely use the faster policy.

Setting these parameters to the optimal modes can boost the I/O performance of guest VMs from <30MB/s on the *aio-stress* disk benchmark to > 800MB/s. This performance increase allows greater observed variability in VM disk throughput when comparing different VM allocation strategies, and therefore is essential for our experiments than if the I/O performance was unnaturally constrained by sub-optimal configuration.

6.3 Alternative workload allocation strategies

In this chapter we evaluate the *Allocator* workload allocation strategy against two of the standard OpenStack allocation algorithms: *spread* and *stack*.

These represent the two traditional approaches to allocating virtual machines in a cloud environment. *Spread* balances the number of VMs across the

available hosts, ensuring no host is overworked and therefore achieving the best possible application performance. Round-robin assignment is an example. *Stack* is the opposite of this strategy, where VMs are *stacked* on the least number of hosts with the aim of reducing the amount of resources required to achieve the body of work.

6.3.1 *spread*

The *spread* algorithm is a load-balancing solution, where the node with the lightest current load is given the new VM request. If there is more than one node with the same level of load or no load at all, then a round-robin system is used to choose the appropriate host. The pseudo code for this algorithm is contained in Snippet 6.4 below.

```
For each host
    Evaluate the load on this host
Choose host with the lightest load.
```

Snippet 6.3: Standard *spread* pseudo code algorithm

The main benefit of this algorithm is improved application performance when compared to other scheduling strategies. *Spread* attempts to balance the workload amongst the available hosts, employing each host at a low level of utilization, reducing VM contention for resources and providing less application degradation than has been seen in highly utilized systems [10].

As this strategy spreads workload amongst a large number of physical hosts it is likely to use a larger amount of energy than alternative strategies. Analysis in the field of energy efficient computing has shown that energy usage of physical machines is directly linked to their consumption of their resources [22]. Some resources cause a larger percentage of that energy consumption, particularly CPU [5], and the lack of energy proportional computers has led to a situation where lightly loaded machines consume a disproportional amount of energy, for example a machine with 10% load consuming more than 60% of the peak energy usage [41].

By employing a large number of lightly loaded hosts, this strategy may incur a higher consumption of energy than an alternative strategy that uses fewer physical machines to conduct its workload.

OpenStack evaluates hosts as “loaded” by how many VMs they have allocated. Because the system does not feature a real-time monitoring system like *CloudMonitor* it cannot accurately assess if the VMs that have been allocated are actually doing any useful work. Therefore OpenStack believes one host with n VMs to be more loaded than another with $n-1$ VMs, regardless of the operations that those VMs are currently conducting. OpenStack treats allocation of resources, rather than their actual use, as loaded or not.

For OpenStack the *spread* algorithm is implemented by first evaluating the allocated memory capacity of each hosts and discarding any that are currently at full capacity. The remaining hosts are then ranked in order of the most available free memory. The host that has the most available free space is the host that OpenStack believes is the least loaded.

The pseudo code algorithm for *spread* as implemented in OpenStack is displayed in Snippet 6.5:

```
For all hosts
    Evaluate currently free RAM
Filter hosts that have free RAM < required for new VM
Sort host by RAM, largest first
Choose host with the largest amount of free RAM
```

Snippet 6.4: Pseudo code algorithm for *Spread* as implemented by OpenStack

6.3.2 *stack*

stack is the opposite placement strategy to *spread* so that VMs are allocated to a single physical host until that host has all of its resources allocated. In this way, hosts are filled before a new host is used.

```
For each host
    Evaluate the load on this host
Choose host with the heaviest load that still has available space
for this VM request.
```

Snippet 6.5: *Stack* pseudo code algorithm

The *stack* strategy is a bin-packing strategy where the aim is to pack as many tasks into single physical hosts or bins as can be achieved. Distribution of the tasks normally means that each physical host is filled to its maximum capacity before another host is employed.

As we have seen that even lightly loaded physical hosts consume significant amounts of energy, lowering the number of physical hosts employed in an allocation strategy is an effective approach for reducing energy consumption.

However, application performance can be severely impacted by this strategy. When a higher number of virtual machines attempt to compete for the physical resources of a host application performance can suffer, as is shown by Srikantiah *et al* [10] and Pinheiro *et al* [42].

As noted above in the *spread* discussion, OpenStack evaluates hosts as “loaded” by how many VMs they have allocated. Therefore, to implement *stack* OpenStack again evaluates each host by how much of their memory (and therefore number of VMs) has been allocated. All host are evaluated in this way and the host that is not yet full, but has enough RAM free to allocate a new VM is chosen.

The pseudo code algorithm for *stack* as implemented in OpenStack is displayed in Snippet 6.7:

```
For the all hosts
    Evaluate currently free RAM
Filter hosts that have free RAM < required for new VM
Sort host by RAM, least first
Choose host with the least amount of free RAM
```

Snippet 6.6: Pseudo code algorithm for *Spread* as implemented by OpenStack

For the experiments described in the next section, the *Allocator* from Chapter 5 was evaluated and compared against both these *spread* and *stack* strategies to determine their effectiveness.

6.4 Experimental results

The *Allocator* software from Chapter 5 was evaluated and compared with the alternative strategies from Section 6.3 at different levels of utilization; first the different workload allocation strategies were evaluated for a load of 25% of the system total capacity, the load was then increased to 50% and the same configurations of work were run again and finally repeated for 75% load. These levels were chosen to represent a variety of system load, while ensuring a manageable number of experiments. 0% and 100% were discarded, as at 0% there would be no load to test and at 100% the differences between algorithms would be negligible, as all hosts would be fully utilized.

The results presented below show the performance and energy consumption of each strategy at these different utilization levels and present a broad look at how a real world system could be affected by implementing one of these approaches.

At each level of utilization, three experiments were conducted, each with different workload configurations. Each workload configuration is made up of 2 of 3 different types of tasks: CPU bound, Memory bound and Disk (DISK) bound. Known as CPU, MEM, and DISK for shortness. The applications used for these benchmarks were the same as Chapter 4.

- The CPU bound task is a benchmark of the *gzip* application – a standard Linux application that can compress files in memory.
- The MEM bound task is a benchmark called *stream* that evaluates the system memory performance.
- The DISK Bound task is a benchmark called *aio-stress* is an asynchronous I/O benchmark created by SuSE. It uses a single thread to consistently read and write a 1024MB test file to and from the hard disk.

6.4.1 25% utilization

The following results are for a VM utilization level of 25%, where 4 VMs were running at any one time on hardware capable of sustaining 16 VMs of the size used for this experiment. VMs were of size 2 CPU cores and 4 GB of RAM. The four physical hosts used in this experiment each had 8 CPU cores and 16GB of RAM making them capable of sustaining 4 VMs each for a total of 16VMs system wide.

For the 25% utilization experiments the system was given, using the *ExperimentRunner* software, request for 4 virtual machines: 2x of one particular type based VMs and 2x of another. When the *Allocator* algorithm was used, each request had the appropriately assigned *scheduler_hint* to identify which type the VM should be.

6.4.1.1 CPU + DISK

This first experiment at the 25% utilization level compares the workload performance and energy usage for a configuration mix of CPU and DISK tasks.

Mapping

The *Allocator* algorithm laid out the tasks by co-locating similar VM types together, so Node A had 2x CPU tasks and Node B had 2x DISK tasks.

For the alternative strategies, *stack* put all four VMs on Node A. *spread*

assigned one task to each of the four nodes.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Mean power draw:	$409 \pm 22.3 \text{ W}^{32}$	$469 \pm 30.5 \text{ W}$	$374 \pm 21.5 \text{ W}$

Table 6.1: Mean power draw by each algorithm in this experiment.

As with the experiments in Chapter 4, the mean power drawn is calculated by examining the energy usage samples at each 3 second interval during the experiment.

The *Allocator* algorithm used more energy than *stack* but less than *spread*. A plot of the variation in power drawn for all three algorithms over the length of this experiment can be seen in the figure below:

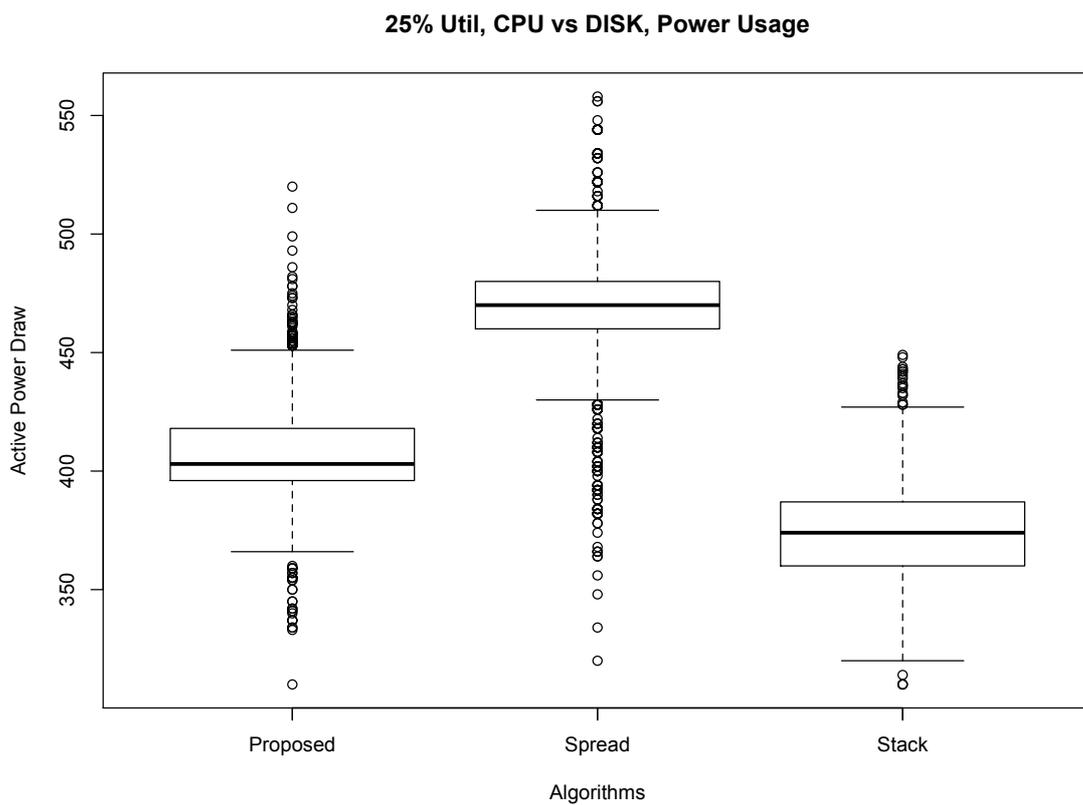


Figure 6.2: Boxplot showing the difference in power drawn by the three algorithms.

³² Type of error used for all results in this chapter was standard deviation.

Over 1,200 power meter readings were taken over the course of this experiment. Analysis of these readings to determine their statistical significance found a p-value of $< 2.2 \times 10^{-16}$ for the *Allocator* algorithm and *stack*, with a mean of the differences of 34.27, and a p-value $< 2.2 \times 10^{-16}$ for the *Allocator* algorithm and *spread*, with a mean of the differences of -60.89. The *Allocator* algorithm power drawn results have a standard deviation of 22.31, meaning both *stack* and *spread* fall farther than one standard deviation away.

The boxplot of Figure 6.2 shows that variance was small for each of the scheduling algorithms. With the small variation in active power draw and the p-value confidence and standard deviation from the statistical analysis above, we can conclude that these results of running this experiment to be statically significant.

The *Allocator* strategy achieved a power saving of 13.3% over the standard *spread* allocation algorithm. *Stack* achieved a 19.9% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>gzip</i> (CPU)	18.77 ± 0.12 s	18.62 ± 0.12 s	18.71 ± 0.09 s
<i>aio-stress</i> (DISK)	363 ± 180.1 MB/s	615 ± 118.9 MB/s	20.58 ± 2.43 MB/s

Table 6.2: Application performance means for this experiment

gzip scores are measured as seconds to compress a 2GB binary file. Therefore a lower score is better. *aio-stress* is measured as I/O throughput per second meaning a higher score is better. In this experiment, the *Allocator* algorithm had the worst CPU performance, but by less than 1%. In contrast, I/O performance compared to the alternative energy-efficient strategy, *stack*, was improved by a factor of 2.

CPU performance for the *Allocator* algorithm was 0.8% worse than *spread* compared to 0.48% for *stack*. Taking I/O performance from *spread* as the benchmark, the *Allocator* algorithm achieved 59% of the I/O throughput, compared to only 3.3% for *stack*.

The standard deviation of the I/O performance was 180MB/s, therefore both *spread* and *stack* fell farther than one standard deviation away. The standard deviation of the CPU performance was 0.12s, therefore *spread* fell farther than one standard deviation away.

6.4.1.2 CPU + MEM

The next experiment at the 25% utilization level compared a CPU+MEM workload configuration for the different allocation algorithms.

Mapping

The algorithms laid out the work identically to previous experiment with the *Allocator* algorithm continuing to group together similar VM types.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Mean power draw:	405 ± 21.8 W	437 ± 25.4 W	395 ± 20.7 W

Table 6.3: Mean power draw by each algorithm in this experiment.

The *Allocator* allocation strategy again required more power than *stack* but less than *spread*. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* allocation strategy achieved an average power saving of 7.3% over the standard *spread* allocation algorithm. *Stack* achieved a 9.6% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>gzip</i> (CPU)	18.68 ± 0.09 s	18.74 ± 0.15 s	19.04 ± 0.19 s
<i>stream</i> (MEM)	7717 ± 785 MB/s	6844 ± 1410 MB/s	6107 ± 625 MB/s

Table 6.4: Application performance means for this experiment

The *Allocator* algorithm had better CPU and MEM performance than both

spread and *stack*. As expected *spread* had better performance than *stack*.

Average CPU performance for the *Allocator* algorithm was 0.3% better than *spread* compared to 1.6% worse for *stack*. Memory performance for the *Allocator* algorithm was a 12.8% improvement over the *spread* algorithm where as *stack* performed 10.8% worse.

Although the magnitude of the standard deviation is high, the mean of the *Allocator* results is more than one standard deviation from mean of both of the alternative strategy results, from which we can conclude that the values are statistically significant.

6.4.1.3 DISK + MEM

The final experiment at 25% utilization level compares the result of the different allocation strategies for a DISK+MEM workload configuration.

Mapping

Mapping of tasks to hosts was again the same for this experiment. The *Allocator* strategy continued to co-locate similar tasks.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Mean power draw:	379 ± 22.4 W	404 ± 24.2 W	372 ± 19.9 W

Table 6.5: Mean power draw by each algorithm in this experiment.

The *Allocator* again used more power than *stack* but less than *spread*. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* achieved a power saving of 6.2% over the standard *spread* allocation algorithm. *Stack* achieved a 7.9% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>Aio-stress (DISK)</i>	1395 ± 123 MB/s	1611 ± 82.6 MB/s	1238 ± 134 MB/s
<i>Stream (MEM)</i>	7421 ± 865 MB/s	6960 ± 1664 MB/s	5982 ± 865 MB/s

Table 6.6: Application performance means for this experiment

The *Allocator* algorithm had better *stream* performance than either *spread* or *stack*, consistent with the results from the previous memory experiment (6.4.1.2). Disk performance for the *Allocator* was between *spread* and *stack*, again consistent with the other disk based experiment (6.4.1.1).

DISK performance for the *Allocator* algorithm was 13.4% worse than *spread* whereas *stack* was 23.2% worse. Memory performance for the *Allocator* algorithm was a 6.6% improvement over the *spread* algorithm whereas *stack* performed 14.1% worse.

6.4.1.4 25% utilization level conclusion

In energy terms, for all three experiments at this level the *Allocator* algorithm used more energy than *stack*, but less energy than *spread*. Energy used for the experiment:

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Energy used for experiments	1188 W/h	1308 W/h	1143 W/h

Table 6.7: Total energy used by each algorithm over the three experiments at this utilization level.

Energy usage statistics from the length of the experiment showed that the *Allocator* algorithm from Chapter 5 used 90.8% of the energy required for the *spread* algorithm, compared to 87.4% for the *stack* placement strategy. The bin-packing strategy saved almost 13% in energy compared with round robin and our strategy competed with a respectable 9% saving for this workload.

- When paired with I/O task, our *Allocator* algorithm had the worst CPU application performance, but the best when paired with memory tasks.
- Memory based tasks had the best performance with the *Allocator*

algorithm under any configuration.

- Disk performance for the *Allocator* algorithm has always better than *stack*, but not better than *spread*.

In conclusion, the *Allocator* algorithm at this utilization level in 5 out of 6 measurement had better application performance than *stack* while still using less energy than *spread*.

6.4.2 50% utilization

The following results are for a VM utilization level of 50%, where 8 VMs were running at any one time on hardware capable of sustaining 16 VMs of the size used for this experiment. VM Sizes remained consistent from the previous utilization level experiments.

For the 50% utilization experiments the system was given, using the *ExperimentRunner* software, request for 8 virtual machines: 4x of one particular type based VMs and 4x of another.

6.4.2.1 CPU + DISK

This first experiment at the 50% utilization level compares the workload performance and energy usage for a configuration mix of CPU and DISK tasks.

Mapping

The *Allocator* algorithm laid out the tasks by co-locating similar VM types together, so Node A had 4x CPU tasks and Node B had 3x DISK tasks, Node C had the remaining DISK task.

For the alternative strategies, *stack* put five VMs (3x CPU and 2x DISK) on Node A, and 3 VMs (2x DISK and 1x CPU) on Node B. *spread* assigned two VMs of varying tasks to each of the four nodes.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	409 ± 22.3 W	469 ± 30.5 W	374 ± 21.5 W
50% power draw:	407 ± 23.1 W	491 ± 38.2 W	415 ± 23.4 W

Table 6.8: Mean power draw by each algorithm in this experiment.

The *Allocator* used less power than *spread* and *stack*. The energy usage for 50% increased from the usage at 25% for both *stack* and *spread*, but *Allocator*'s energy usage was reduced. The same statistical analysis was conducted as the previous experiments and found the results to be statistically significant.

The *Allocator* allocation strategy achieved a power saving of 17.1% over the standard *spread* allocation algorithm. *Stack* achieved only a 15.5% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>Gzip</i> (25%)	18.77 ± 0.12 s	18.62 ± 0.12 s	18.71 ± 0.09 s
<i>Gzip</i> (50%)	19.04 ± 0.21 s	18.74 ± 0.18 s	18.74 ± 0.15 s
<i>Aio-stress</i> (25%)	363 ± 180.1 MB/s	615 ± 118.9 MB/s	20.58 ± 2.43 MB/s
<i>Aio-stress</i> (50%)	1250 ± 159 MB/s	1450 ± 128 MB/s	1112 ± 374 MB/s

Table 6.9: Application performance means for this experiment at 50% and 25% utilization

The *Allocator* algorithm had the worst CPU performance by a small margin, but better I/O performance than *stack*. Compared to the 25% utilization level, the 50% results were better for all DISK usage and slightly worse over all for CPU.

CPU performance for the *Allocator* algorithm has 1.6% worse than *spread* that was the same as *stack*. Taking I/O performance from *spread* as the benchmark, the *Allocator* algorithm achieved 86.2% of the I/O throughput, compared to only 73.1% for *stack*.

6.4.2.2 CPU + MEM

This next experiment at the 50% utilization level compares the workload performance and energy usage for a configuration mix of CPU and MEM tasks.

Mapping

Mapping of tasks to hosts was again the same for this experiment. The *Allocator* strategy continued to co-locate similar tasks.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	405 ± 21.8 W	437 ± 25.4 W	395 ± 20.7 W
50% power draw:	409 ± 29.1 W	420 ± 84.7 W	419 ± 65.1 W

Table 6.10: Mean power draw by each algorithm in this experiment.

The *Allocator* used less power than both *stack* and *spread*. Energy usage from the 25% to 50% level increased for all algorithms except *spread*. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* achieved a power saving of 2.6% over the standard *spread* allocation algorithm. *Stack* achieved only a 0.2% reduction in power usage.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>Gzip</i> (25%)	18.68 ± 0.09 s	18.74 ± 0.15 s	19.04 ± 0.19 s
<i>Gzip</i> (50%)	18.74 ± 0.11 s	18.67 ± 0.15 s	18.81 ± 0.16s
<i>stream</i> (25%)	7717 ± 785 MB/s	6844 ± 1410 MB/s	6107 ± 625 MB/s
<i>stream</i> (50%)	7712 ± 856 MB/s	6103 ± 1008 MB/s	7085 ± 826 MB/s

Table 6.11: Application performance means for this experiment

The *Allocator* algorithm had better MEM performance than both *spread* and

stack. CPU performance was best for *spread*, then our *Allocator* then *stack*. Interestingly, the worst memory performance at 50% utilization was on the *spread* strategy, something that is consistent with the next experiment (6.4.2.3) as well.

Compared to the 25% utilization level, the 50% results were worse for *spread*, but better for *stack*. The *Allocator* algorithm managed similar *stream* performance but worse *gzip*. Both results were very close, suggesting that the algorithm handled the increase in utilization well.

CPU performance for the *Allocator* algorithm was 3.7% worse than *spread* compared to 7.5% worse for *stack*. Memory performance for the *Allocator* algorithm was a 26.4% improvement over the *spread* algorithm and *stack* was 16.1% better.

6.4.2.3 DISK + MEM

The final experiment at 50% utilization level compares the result of the different allocation strategies for a DISK+MEM workload configuration.

Mapping

Mapping of tasks to hosts was again the same for this experiment. The *Allocator* strategy continued to co-locate similar tasks.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	379 ± 22.4 W	404 ± 24.2 W	372 ± 19.9 W
50% power draw:	402 ± 28.9 W	421 ± 31.0 W	403 ± 25.6 W

Table 6.12: Mean power drawn by each algorithm in this experiment.

Again at 50% utilization, the *Allocator* drew less power than both *stack* and *spread*. Energy usage from the 25% to 50% level increased for all algorithms. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant for the *spread* vs the *Allocator* algorithm but for because the power values for *stack* and the *Allocator*

allocation strategy were very close with a mean of the differences of only - 0.175, the difference between them was not statistically significant and gave a p-value of 0.8479.

The *Allocator* had the lowest energy usage at 4.5% less than *spread*. *Stack* managed a 4.3% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>aio-stress</i> (25%)	1395 ± 123 MB/s	1611 ± 82.6 MB/s	1238 ± 134 MB/s
<i>aio-stress</i> (50%)	1359 ± 163 MB/s	1539 ± 132 MB/s	1339 ± 135 MB/s
<i>stream</i> (25%)	7421 ± 865 MB/s	6960 ± 1664 MB/s	5982 ± 865 MB/s
<i>stream</i> (50%)	7085 ± 826 MB/s	6517 ± 912 MB/s	7142 ± 899 MB/s

Table 6.13: Application performance scores for this experiment

The *Allocator* algorithm had disk performance between *spread*, the best, and *stack*. Memory performance on *spread* was again poor at 50% utilization, with both the *Allocator* algorithm and *stack* bettering its throughput.

DISK performance for the *Allocator* algorithm was 11.7% worse than *spread* compared to 13.0% worse for *stack*. Memory performance for the *Allocator* algorithm was an 8.7% improvement over the *spread* algorithm whereas *stack* managed a 9.6% improvement.

Compared to the 25% utilization level, the 50% results were worse for *spread*, but better for *stack*. The *Allocator* algorithm results were also slightly worse. Both results were very close, suggesting that the algorithm handled the increase in utilization well.

6.4.2.4 50% utilization level conclusion

In energy terms, for all three experiments at this level of utilization the *Allocator* algorithm used less energy than both of the alternative strategies.

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Energy used for experiments	1223 W/h	1366 W/h	1245 W/h

Table 6.14: Total energy used by each algorithm over the three experiments at this utilization level.

Energy usage statistics from the length of the experiment showed that the *Allocator* algorithm from Chapter 5 used 89.5% of the energy required for the *spread* algorithm, compared to 91.1% for the *stack* placement strategy. The bin-packing strategy saved almost 8.9% in energy compared with load balancing strategy and our *Allocator* bettered this with a 10.5% saving overall.

- When paired with I/O task, our *Allocator* algorithm again had the worst CPU application performance, as was seen at 25% utilization. CPU performance was better than *stack* when paired with memory tasks.
- Memory based tasks always had better performance than *spread* in but were beaten by *stack* in one situation.
- Disk performance for the *Allocator* algorithm was always better than *stack* but not better than *spread*.

In conclusion, the *Allocator* algorithm at this utilization level had better application performance than *stack* in 4 out of 6 measurements and used the least amount of energy of any of the allocation strategies. A hypothesis for these performance results could be that due to the nature of *stack* and *spread* not evaluating their workload types before VM placement they sometimes were able to place workload in an optimal way that would improve application performance. This however was not done consistently, unlike the performance of the *Allocator* that always takes this into consideration. The *Allocator* is attempting to balance performance and energy and these results would suggest that it leans towards energy reduction more than performance optimisation.

Using the least amount of energy here is notable, because for all of the experiments, *stack* would consolidate the VMs onto just two hosts, leaving two idle. In each case, our proposed solution used 3 or even 4 servers to complete the workload but still managed to use less energy overall because each host

was working at a lighter load and *stack* was unable to power down unused hosts.

6.4.3 75% utilization

The following results are for a VM utilization level of 75%, where 12 VMs were running at any one time on hardware capable of sustaining 16 VMs of the size used for this experiment. VM sizes remained consistent from the previous utilization level experiments.

For the 75% utilization experiments the system was given, using the *ExperimentRunner* software, request for 12 virtual machines: 6x of one particular type based VMs and 6x of another.

6.4.3.1 CPU + DISK

This first experiment at the 75% utilization level compares the workload performance and energy usage for a configuration mix of CPU and DISK tasks.

Mapping

The *Allocator* algorithm laid out the tasks by co-locating similar VM types together, so Node A had 6x CPU tasks and Node B had 3x DISK tasks, Node C had the remaining 3x DISK tasks.

For the alternative strategies, *Stack* put five VMs (3x CPU and 2x DISK) on Node A, five VMs (3x DISK and 2x CPU) on Node B and the two remaining VMs on Node C. *Spread* assigned three VMs of varying tasks to each of the four nodes.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	409 ± 22.3 W	469 ± 30.5 W	374 ± 21.5 W
50% power draw:	407 ± 23.1 W	491 ± 38.2 W	415 ± 23.4 W
75% power draw:	449 ± 37.1 W	499 ± 51.2 W	456 ± 44.6 W

Table 6.15: Mean power draw by each algorithm in this experiment.

The *Allocator* used less power than both *stack* and *spread*. Energy usage from the previous levels increased for all algorithms. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* had the lowest energy usage at 10.0% less than *spread*. *Stack* managed an 8.6% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>Gzip</i> (25%)	18.77 ± 0.12 s	18.62 ± 0.12 s	18.71 ± 0.09 s
<i>Gzip</i> (50%)	19.04 ± 0.21 s	18.74 ± 0.18 s	18.74 ± 0.15 s
<i>Gzip</i> (75%)	19.06 ± 1.25 s	18.89 ± 0.28 s	18.83 ± 0.17 s
<i>Aio-stress</i> (25%)	363 ± 180.1 MB/s	615 ± 118.9 MB/s	20.58 ± 2.43 MB/s ³³
<i>Aio-stress</i> (50%)	1250 ± 159 MB/s	1450 ± 128 MB/s	1112 ± 374 MB/s
<i>Aio-stress</i> (75%)	1244 ± 134 MB/s	1375 ± 177 MB/s	1203 ± 390 MB/s

Table 6.16: Application performance scores at all levels for this experiment

The *Allocator* algorithm had the worst CPU performance by a small margin, but better I/O performance than *stack*. The CPU and I/O performance of the

³³ Aio-Stress results at 25% are significantly lower than the levels due to optimisations in the software that were conducted after the 25% experiments ran. As we are only comparing the mixes within each utilization level, these results are still valid.

spread algorithm fell from the 50% utilization rates when the *Allocator* strategy and *stack* did not suffer the same degradation.

CPU performance for the *Allocator* algorithm was 0.9% worse than *spread* compared to the same *stack*. Taking I/O performance from *spread* as the benchmark, the *Allocator* algorithm achieved 90.5% of the I/O throughput, compared to only 87.5% for *stack*.

CPU performance all strategies decreased from the previous level. DISK performance for *spread* and the *Allocator* was worse, but *stack* improved slightly. The *Allocator* DISK performance was only 0.4% worse despite the increased load.

6.4.3.2 CPU + MEM

This next experiment at the 75% utilization level compares the workload performance and energy usage for a configuration mix of CPU and MEM tasks.

Mapping

Mapping of tasks to hosts was again the same for this experiment. The *Allocator* strategy continued to co-locate similar tasks.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	405 ± 21.8 W	437 ± 25.4 W	395 ± 20.7 W
50% power draw:	409 ± 29.1 W	420 ± 84.7 W	419 ± 65.1 W
75% power used:	461 ± 42.0W	516 ± 51.6 W	494 ± 49.5 W

Table 6.17: Mean power draw by each algorithm in this experiment.

The *Allocator* used less power than both *stack* and *spread*. Energy usage from the previous levels increased for all algorithms. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* had the lowest energy usage at 10.6% less than *spread*. *Stack* managed a 4.3% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>gzip</i> (25%)	18.68 ± 0.09 s	18.74 ± 0.15 s	19.04 ± 0.19 s
<i>gzip</i> (50%)	18.74 ± 0.11 s	18.67 ± 0.15 s	18.81 ± 0.16s
<i>gzip</i> (75%)	19.4 ± 1.05 s	18.95 ± 0.26 s	19.00 ± 0.25 s
<i>stream</i> (25%)	7717 ± 785 MB/s	6844 ± 1410 MB/s	6107 ± 625 MB/s
<i>stream</i> (50%)	7712 ± 856 MB/s	6103 ± 1008 MB/s	7085 ± 826 MB/s
<i>stream</i> (75%)	6313 ± 1073 MB/s	6719 ± 983 MB/s	6433 ± 653 MB/s

Table 6.18: Application performance scores for this experiment

The *Allocator* algorithm had the worst CPU performance by a small margin, but better MEM performance than *spread*.

CPU performance for the *Allocator* algorithm was 2.4% worse than *spread* compared to 2.6% worse for *stack*. The *Allocator* algorithm had a 6.0% reduction from the *spread* I/O throughput, while *stack* had a 4.3% reduction.

CPU performance for all algorithms fell from the previous levels. MEM performance for *spread* improved but the others fell.

6.4.3.3 DISK + MEM

The final experiment at 75% utilization level compares the result of the different allocation strategies for a DISK+MEM workload configuration.

Mapping

Mapping of tasks to hosts was again the same for this experiment. The *Allocator* strategy continued to co-locate similar tasks.

Power usage

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
25% power draw:	379 ± 22.4 W	404 ± 24.2 W	372 ± 19.9 W
50% power draw:	402 ± 28.9 W	421 ± 31.0 W	403 ± 25.6 W
75% power used:	440 ± 34.4 W	460 ± 40.4 W	432 ± 35.7 W

Table 6.19: Mean power draw by each algorithm at each level of this experiment.

The *Allocator* used less energy than *spread* but more than *stack*. Energy usage from the previous levels increased for all algorithms. The same statistical analysis was conducted as the previous experiment and found the results to be statistically significant.

The *Allocator* had energy usage at 4.3% less than *spread*. *Stack* managed a 6.1% reduction.

Application performance

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
<i>aio-stress</i> (25%)	1395 ± 123 MB/s	1611 ± 82.6 MB/s	1238 ± 134 MB/s
<i>aio-stress</i> (50%)	1359 ± 163 MB/s	1539 ± 132 MB/s	1339 ± 135 MB/s
<i>aio-stress</i> (75%)	1253 ± 111 MB/s	1440 ± 150 MB/s	1200 ± 278 MB/s
<i>stream</i> (25%)	7421 ± 865 MB/s	6960 ± 1664 MB/s	5982 ± 865 MB/s
<i>stream</i> (50%)	7085 ± 826 MB/s	6517 ± 912 MB/s	7142 ± 899 MB/s
<i>stream</i> (75%)	5827 ± 664 MB/s	7224 ± 732 MB/s	5668 ± 825 MB/s

Table 6.20: Application performance scores for this experiment

The *Allocator* algorithm had disk performance between *spread*, the best, and *stack*. Memory performance was similarly between *spread* and *stack*.

DISK performance for the *Allocator* algorithm was 13.0% worse than *spread* compared to 16.7% worse for *stack*. In this experiment, the *Allocator* algorithm had 19.4% worse MEM performance than the *spread* algorithm. *Stack* was

21.5% worse.

Performance across both applications on all algorithms fell from the previous level, except Memory performance on *spread*.

6.4.3.4 75% utilization level conclusion

In energy terms, for all three experiments at this level of utilization the *Allocator* algorithm used less energy than both of the alternative strategies.

	<i>Allocator</i>	<i>Spread</i>	<i>Stack</i>
Energy used:	1349 W/h	1517 W/h	1390 W/h

Table 6.21: Total energy used by each algorithm over the three experiments at this utilization level.

Energy usage statistics from the length of the experiment showed that the *Allocator* algorithm from Chapter 5 used 88.9% of the energy required for the *spread* algorithm, compared to 91.6% for the *stack* placement strategy. The bin-packing strategy saved almost 8.3% in energy compared with load balancing strategy and our proposed system bettered this with an 11.1% saving overall.

When paired with I/O task, our *Allocator* algorithm again had the worst CPU application performance, as was seen at 25% and 50% utilization.

MEM + Disk performance for the *Allocator* algorithm was always better than *stack* but not always better than *spread*. This was the same situation as 50% utilization.

6.5 Conclusions

These experiments aimed to evaluate the effectiveness of the *Allocator* workload allocation strategy detailed in Chapter 5. Alternative strategies for reducing energy consumption in private cloud computing systems have focused on consolidation, which while reducing the total amount of energy required, has impacts on application performance.

In these experiments, unlike in Chapter 4, there was an absence of network

based performance tests. This was due to unforeseen issues when running a number of network intensive tasks on the *nova-network* software stack. Often the stack would crash, rendering VMs unresponsive and the experiment void. Future work for this *Allocator* should incorporate a network performance assessment that could be run at scale without impacting the *nova* architecture. However, our results in Chapter 4 suggest that CPU and NET jobs place similar loads on the hardware so we are confident in presenting these results without individual attention on NET tasks.

Our strategy showed average power savings between 4 and 19% depending on the workload, whereas *stack*, the alternative energy efficient algorithm achieved reductions of between 2 and 20% also dependent on workload.

Average saving at each level:

- 25%
 - *Stack* strategy: 13%
 - *Allocator* strategy: 9%

- 50%
 - *Stack*: 8.9%
 - *Allocator*: 10.5%

- 75%
 - *Stack*: 8.3%
 - *Allocator*: 11.1%

The average saving at each level for the *Allocator* algorithm was around 10%. The alternative strategy started out at 13% better and fell to 6.4% at the higher levels of utilization. For each level our algorithm reduced energy consumption whereas *stack* increased its consumption.

The results of improvement may be small in terms of the amount of energy consumed, but at the data centre scale, where OpenStack is frequently

deployed, an energy reduction of 8.3% per year would be significant, especially without a large application performance penalty.

The *Allocator* algorithm exploits the characteristics of workload mixes to reduce energy usage for certain workloads by around 10%. For a private cloud installation of 10,000 machines, as discussed in the Chapter 1 introduction, that spends £1.56 million per year on electricity, there would be potentially £156,000 in energy savings each year by implementing the *Allocator* algorithm. These costs do not take into account any reduction in associated infrastructure expenditure such as air conditioning. The nature of workload mixes leading to less contention of resources may also lead to reductions in these associated costs, if for example the overall machine temperatures are lower thanks to the workload mix.

At the 25% utilization level, the *Allocator* always saved energy compared to the standard load balancing strategy, around 10% reduction overall for the experiments at this level compared to 13% for the alternative energy saving strategy. At 25% the *Allocator* algorithm also had the best MEM performance and better DISK performance than *stack*.

For 50% utilization, the *Allocator* algorithm always used the least amount of energy and had better performance than *stack* in 4 out of 6 application benchmarks. Interestingly in these situations *stack* attempted to consolidate all VMs onto 2 hosts, where our approach would use 3 or 4 hosts for the same workload. Despite using more servers, our algorithm always had the lowest energy consumption of the three tested at this level, suggesting that it is possible to reduce energy consumption without concentrated consolidation.

At 75% the energy savings of the *Allocator* remained consistent at approximately 10% where as *stack*'s fell to 6.4%. This extra energy expended by *stack* seemed to result increased application performance as it scored better CPU results than the *Allocator* for the first time. However, the *Allocator* continued to outpace *stack* in the other performance benchmarks MEM and DISK.

At the higher levels of utilization, the *Allocator* algorithm generally had the worst CPU results, but not normally by more than 1 or 2% of the best possible

result. Notably, at these higher levels of utilization the *Allocator* generally had the lowest energy consumption levels, suggesting that in a highly utilized system, lowering CPU performance by just 1% can have big impacts on the total amount of energy used. This would probably be undetectable for the vast majority of users and would be consistent with observations in the field that CPU has the biggest impact on energy performance.

The *Allocator* algorithm may have been too focused on saving energy, which led to detrimental impacts to CPU performance. The design aim of the *Allocator* was to reduce energy while maintaining application performance. In some scenarios the energy savings for *Allocator* were higher than *stack*, the supposedly most energy efficient algorithm. In future work, the behaviour of the *Allocator* could be adjusted to improve application performance at higher levels of utilization, which would likely come at the cost of using an increased amount of energy.

Stack power results would have had a much larger energy saving if their unused machines were powered down. The results of this experiment are based on the scenario that assumes that, for reasons of dependability, latency and complexity, organisations do not power down unused resources and in such a situation the *Allocator* algorithm would save more energy at 50% and 75% utilization levels.

The experiments in this chapter were all conducted using type 2, full virtualization (KVM on Ubuntu hosts). If a different virtualization technique was employed then the results in this chapter may differ. Type 1 virtualization would employ no underlying host operating system, which would reduce the total amount of resources consumed by each host.

Paravirtualization guest VMs are aware of other guests on the same host and use that information to tailor their hardware requests as a cohesive whole system. This would likely lead to situations where application performance is degraded to avoid overwhelming the host. However as full virtualization emulates a full physical machine to each guest the hypervisor cost is higher than in para-virtualization. Both modes would therefore have different impacts on performance and energy consumption and the magnitude of that impact could be investigated as future work.

It is possible to make hypotheses about why the results of these experiments are as they are, but such claims would be outside the scope of this work due to complexity of the underlying software and hardware system interactions between applications, guest OS, hypervisor, host OS and components. However, future experiments to detect the reasons and understand the complexity of these interactions would be useful in designing power efficient hardware and improving the energy efficiency of virtualized systems.

These experiments are based on the assumption that one virtual machine has one application of the type described operating within it. In another case where multiple applications were running inside a single VM, these results would need to be re-investigated.

The work done in this project was limited to experiments on homogeneous hardware due to the practicalities and restrictions of our research environment. The experiments evaluated show that *Allocator* does provide a different solution to the alternative strategies for these workloads on this hardware. It is conceivable that the conclusions drawn here could differ if different hardware was used, but it is logical that there would be some effect of workload mixing on energy usage and application performance because we believe that the work is founded on strong basic principles. It is likely that any new hardware would need to be evaluated to uncover the mix effects and then the *Allocator* software could be adjusted accordingly to take these effects into account.

The experiments in this work compare the *Allocator* software to the OpenStack implementation of *spread* and *stack*, which are particularly influenced by the OpenStack definition of a *loaded* host. OpenStack specifies a host as loaded if its resources have been allocated to a virtual machine, regardless if that VM is actually using those resources at the current time. If a different definition of *loaded* was prepared for OpenStack, where VMs could be allocated based on actual use of resources rather than “allocated” resources then a new set of experiments would be required.

These results demonstrate that workload mixes are capable of reducing energy consumption while limiting application performance penalties. The *Allocator* shows that an implementation of the mixes is possible in an OpenStack

compute environment. The individual mixes that are used and the placement characteristics that they exploit are particular to this hardware and software configuration and would require additional experimentation if a component was modified.

However, some results could be generalized. It would not be unfair to say that today's computers have become exceedingly efficient at maximising performance from CPU sharing such that placing CPU jobs together will have little impact on application performance. Placing these high-power consuming tasks together reduces impact on the other applications and consigns the high-energy jobs to a concentrated number of hosts. Additional work in this area could focus on developing a set of such generalized rules that could be deemed best practices for allocation workload in computing systems.

7 Conclusions and future work

This thesis has presented two software tools: *CloudMonitor*, which aims to help organizations instrument their energy consumption at scale, and an OpenStack *Allocator* that can reduce the amount of energy used by a private cloud installation, while limiting the impact to application performance when compared to alternative energy-efficient allocation strategies.

CloudMonitor uses a power model to estimate the energy usage of a system after it has been trained against an accurate hardware power measurement device. A power model is a mathematical expression that attempts to capture the relationship between subcomponent use (such as CPU, DISK, etc) and energy consumed by the machine. Hence measurement of hardware utilization allows the energy consumed to be calculated.

As organizations move their internal infrastructure solutions to a private cloud solution, conflicting interests arise. Organizations are determined to reduce energy consumption, either motivated by the rising cost of fuel or through efforts tackle climate change and become a sustainable institution. At the same time, they want to take advantage of the benefits given by new technologies such as cloud computing. The challenge is to exploit the new technology while attempting to reduce their overall energy consumption.

The work done in this thesis represents efforts to help organizations combat this challenge. Most organizations do not fully understand their overall energy usage or the energy profile of their IT infrastructure. This lack of understanding is the main barrier to adoption of energy efficient computing techniques, particularly when new technology such as cloud computing is involved. My work has developed tools to aid this understanding and exploit the characteristics of private cloud computing to reduce energy usage.

Our work had two broad objectives, which were accomplished during the course of this thesis:

1. *Develop a scalable mechanism for measuring the energy consumption of shared servers in a cloud environment.*

After surveying the literature in the area of energy measurement, an

application was developed to aid organizations in their understanding of their energy consumption.

The *CloudMonitor* software comprises a resource monitor, direct tool for interfacing with power measurement hardware and an energy estimation tool that uses power models to accurately estimate energy usage.

CloudMonitor can help an organization measure their energy usage by providing a means of gathering that information at scale, without adding an external power meter to each device. Unlike other solutions in the field, it is open source, portable and calculates the power model required automatically during the training phase. We are confident that *CloudMonitor* is accurate as it has been trained against a billing level hardware power measurement device and has shown in our experiments to have an error rate of around 4%. Such an error rate is consistent with, or slightly better than other power models in the literature.

- 2. Build a virtual machine allocation strategy for a private cloud system that reduces energy usage without compromising application performance or dependability.*

As more organizations are turning to private cloud solutions for their IT needs, generally driven by security and operational concerns, the lack of an appropriate energy efficient workload allocation system for OpenStack, one of the most popular private cloud platforms, becomes a challenge with increasing potential to positively impact and reduce the energy consumption of organisations.

Traditional approaches to reducing energy costs through workload allocation have focused on consolidation [14][15][16][17], sometimes to the detriment of application performance. My work developed an energy-efficient allocation strategy for OpenStack that also attempted to preserve application performance. It was found that the solution achieved around 10% energy savings at all levels of utilization, and achieved generally better application performance results than the alternative, consolidation-focussed, energy efficient allocation strategy.

Overall, these software applications help to enable organizations to understand their energy consumption and take effective steps to reduce that consumption on their internal IT infrastructure.

7.1 Critical evaluation of the work done

The *CloudMonitor* software attempts to move forward the current work in software energy estimation by automatically evaluating a power model against billing level accuracy power meters to create a tailored model that closely follows the relationship between system subcomponents and overall energy usage. The power model is generated once during the training phase and is not automatically adapted over the lifetime of the system.

This could be improved by a feedback loop or neural network that constantly adjusts the power model to give the best results for the current hardware performance. The evaluation of *CloudMonitor* in this thesis shows that when trained correctly the power model can have up to 96% accuracy when compared to the actual energy consumption figures. Additional, continuous feedback could increase this accuracy.

Introducing any monitoring platform into a system can lead to the emergence of the *observer effect*, where the monitoring impacts the performance of the system. To limit this effect *CloudMonitor* is written in a way to reduce computational processes to the only those minimum that are required to complete the task of metering and recording values in the system. The SIGAR library used as the resource usage collection mechanism has been *streamlined* to reduce load, and *CloudMonitor* is also tailored to reduce the number of data transfer commands required and therefore minimize any additional overhead.

Current academic work in developing an energy efficient VM allocation system for OpenStack is limited to reducing the number of physical machines in use through consolidation. There is little work that focuses on maintaining application performance or evaluating systems that do not power down unused resources. The *Allocator* that has been developed in this work addresses some of these challenges by presenting an approach that does

balance performance needs with the desire for a reduction in energy costs.

The list of alternative strategies to the developed *Allocator* algorithm are as follows:

- *stack* (generic consolidation algorithm): implemented for our evaluation experiments, *stack* attempts to utilize as few physical machines as possible in order to reduce energy without consideration of application performance.
- *spread* (generic load-balancing algorithm): implemented for our evaluation experiments, *spread* attempts to balance load evenly between the available physical hosts without consideration of energy usage but usually resulting in the best application performance thanks to lightly loaded hosts and the lack of VM contention for resources.
- Liao [14] describes a consolidation algorithm that gives respect to application SLAs. The energy savings of such an approach could be significant but are dependent on the SLA set. Application performance is only guaranteed to meet this level, and energy savings are therefore variable.
- Corradi [15] present a basic consolidation algorithm that gives no heed to application performance, it is most similar to the *stack* algorithm described above.
- Beloglazov [16] describe a migration based consolidation algorithm but do not present the finished implementation for discussion or comparison.
- Lindgren [17][78] also detail a placement system that can effectively load balance VMs, but do not present an analysis of their algorithm in terms of energy saved or application performance.

A table summarising these approaches and comparing it with the *Allocator* software is presented in Table 7.1.

The *Allocator* does also not employ live migration or reactionary allocation of

VMs. This could lead to situations where the system is not in the most optimal workload configuration for either performance or energy consumption. However, the benefits of a system without migration of VMs should be more stability and the removal of overhead costs of those migrations in both system availability and increased resource consumption. This was seen as the most beneficial situation for the overall system if the initial *Allocator* was sufficiently effective in its decisions.

The developed *Allocator* for OpenStack relies upon accurate input from users to properly classify the workloads that are running on the system. If the user is untruthful or mislabels a VM request, then performance and energy consumption on the system will be impacted. Properly labelling VM requests may be difficult for some users if they are unaware of the resources that their tasks require. Running the *CloudMonitor* software alongside any task and using its profiling ability to determine the appropriate type for subsequent runs could overcome this lack of knowledge.

However, end-users are incentivized to be accurate with their request classifications, as doing so correctly should result in the best possible performance for their applications. In an alternative system, where the focus is only on reducing energy costs, users would notice that the performance of their applications becomes degraded and would therefore not be as motivated to conform to the new requirements placed on them requiring organizational pressure to ensure compliance. When application performance preservation is promised along with savings in energy, users should be suitably incentivised to comply with the labelling requirements.

Authors	Approach	Application Performance	Energy Savings	OpenStack Version
Liao [14]	Consolidate with respect to SLAs	Only sufficient to meet SLAs	Significant, but figures not provided and unable to read from graph	Not Provided
Corradi [15]	Consolidation	Not provided	Very basic analysis, based on consolidation. Example given is if VMs from 5 machines can be consolidated to 1 then 80% reduction in energy is achieved.	Diablo
Beloglazov [16]	Live Migration consolidation	No evaluation	No evaluation	Not provided
Lindgren [17][78]	Hybrid initial placement/live migration system. Either consolidated or balanced	CPU utilization maximum was never more than 10% from the average, but no figures given of application performance.	Not provided except to say that no more 1 host more than was needed was used.	Diablo
Smith	Initial placement according to workload mix strategy	Attempts to preserve application performance from by collocating compatible tasks.	Approximately 10% reductions in energy consumption when compared to a standard load balancing algorithm. Better than consolidation algorithms at higher levels of utilization	Essex

Table 7.1: Comparative table of approaches to modifying OpenStack, including *Allocator*

Without compliance, users could mislabel workload and therefore reduce the effectiveness of the system. VMs would be allocated in the wrong place and application performance could suffer along with an increase in overall system energy usage. In order to avoid this situation, system administrators could run *CloudMonitor* within each VM to monitor the used resources and report any

mislabelling back to the users.

Some bin-packing VM allocation strategies presented in the literature claim energy savings of up to 30% at the cost of impacted application performance. Our evaluation implemented a bin-packing strategy that at best achieved 13% energy savings. The disparity in these reductions could be explained by our test bed not powering down unused machines, for example for the 75% CPU+DISK experiment, if *stack* powered down the unused server the energy reduction would have been 28% rather than the 9.4% it achieved. The *Allocator* algorithm (11.7% reduction) therefore achieves similar energy usage when the unused idle machines are still powered on and consuming electricity.

In this situation, which the literature suggests is common within organizations, the work in this thesis presents an improvement in the field. Turning off machines in data centres is normally impractical due to the associated latency delays from power down and power up transitions, increased hardware failure rates due to power cycling and additional complexity of the software required to deal with inconsistent hardware.

A potential weakness in the experiments is based around the workloads chosen to represent each of the VM types. For example, while most CPU intensive tasks are broadly similar and therefore the effects of a different CPU task would be minimal, some of the other subcomponents may have different usage patterns that could impact the effectiveness of these experiments. An example of this could be the I/O intensive operations can be *consistent* or *lumpy* depending on their workload characteristics. The test used in the experiments, *aiio-stress*, is a consistent stream of hard disk reads and writes, while an alternative application may have a different profile and so a different end result for the experiments.

Therefore, it is important to note that the results of these experiments, while we believe them to be logical and sound, are dependent on these workloads on the configuration of hardware used. It is conceivable that workloads with different characteristics would have a different effect. However, we believe from the work in this thesis that some effect would be present regardless of hardware or workload configuration. The nature of that effect could be harnessed and adapted in the *Allocator* software for it to be effective on the

new configuration.

The *Allocator* that was designed and developed in this thesis is a prototype of how such software could be used in a production system. Therefore it relies upon deployment-specific heuristics, such as the hardcoded values for the *Weigher* module. In order for the software and ideas presented in this thesis to be adapted to a different situation, changes to the software would be required.

Firstly, the software should be edited to be more easily customisable to the deployment environment. One way in which this could be achieved would be to make sure that all deployment specific values and directions are extracted to a configuration file, allowing the system administrator to tailor the software and workload mix to their system.

Future implementations may also benefit from using the real time information provided by *CloudMonitor* to create a feedback loop that adjusts the workload mix strategies used depending on the actual performance metrics of the software and energy consumed during work. Such a system would not only benefit from being able to be deployed anywhere but would also make the solution expandable to heterogeneous deployments and changes to software configurations, without re-coding of the mix parameters.

7.2 Discussion

During the course of this work the current state of the art in energy-efficient VM allocation algorithms were evaluated and found have a significant impact on application performance. The majority of these algorithms favour a consolidation-based approach, where VMs are consolidated on to the minimal number of physical machines. In these situations the physical host achieves a higher level of utilization and the VMs experience increased contention for resources.

If a strategy that relies upon machines being powered down to reduce energy load is deployed in a situation where machines are kept on all of the time, as the literature suggests is common due to the associate latency delays,

increased hardware failure rates and additional software complexity, that strategy will not produce an optimal energy situation.

In the future, computing server hardware may improve its resilience to power cycles and reduce time to switch between power states. If this occurs, then the practical case for use of switch-off consolidation algorithms will be stronger.

In the search for an alternative to these switch-off consolidation algorithms, we found that different arrangements of the same workload on fixed hardware could have an impact on both energy consumption and application performance. Given these observations, there is a possibility of an optimal workload mix for any given set of workload on any hardware. Using these insights this project aimed to develop a sufficiently energy efficient *Allocator* for OpenStack that preserved application performance.

7.3 Future work

Experiences with the software and experiments used in this work lead to a number of possible areas that could be improved in the future.

To further evaluate the work done on the *Allocator* algorithm, it might be appropriate to replace the synthetic workload generator used in Chapter 6 to evaluate the workload allocation mechanisms with one based on a real cloud workload trace. Such a trace could improve the experiment's correlation with real world activities and further validate our approach. To conduct a trace experiment, first data from the use of a real world cloud platform would have to be obtained, then that trace would need to be parsed to influence the synthetic workload generator tool. The tool would follow the trace, launching and terminating VM instances to mimic the period from the real world workload.

Additional work could also be done to evaluate the impact of different virtualization types on workload mixes. The mixes described in this thesis are the result of experimentation on type 2, full virtualization, but experiments on type 1 and paravirtualization systems would also be worthy of consideration.

The results of the experiments in this thesis allowed certain heuristics to be devised that were applicable to the hardware and software configurations used. To determine if these results could be generalized, additional experimentations could be conducted on different hardware hosts and software configurations. In order to ascertain if these heuristics generalise the experiments presented in Chapter 4 would need to be conducted on different platforms. The number of platforms that would be tested could be dependent on the variation between the experiment results on the initial platforms that are examined.

There are also interesting research areas that could lead to the *Allocator* software been expanded to become more sophisticated in its allocation decisions. If a private cloud system was deployed on heterogeneous hardware with different chipsets and internal components, each machine could be evaluated to determine its suitability to different combinations of workload. In such a situation, in which the *Allocator* would have access to this knowledge, decisions could be taken to assign workload specifically to the host with the hardware that is most suitable to the work to be undertaken. In this way the *Allocator* would no longer treat each host as equal, which would lead to some interesting scheduling challenges.

To enable this *Allocator*, the private cloud user was asked to provide additional meta data about the VM instance they are requesting. The users provided a *type* to describe their VM and the resources it would consume, from one of CPU, MEM, DISK or NET. Prompting user to provide additional meta-data about the jobs they want to run opens up a large array of potential energy saving options.

If, in future work, the user could describe in greater detail the work to be performed then the *Allocator* could tailor decisions even closer to the optimal energy usage / performance situation. The act of users assigning meta-data to their VM instance request could be automated going forward, with the system recognizing a job profile from an internal database. Either through this kind of automation or by simply asking users to provide more information there is potential scope for improving the user experience and further optimizing the private cloud platform.

In such a situation, the *Allocator* would request more information from the user, which may be asked to mark-up their job in a suitable language like XML. The ideal situation would be that each job would be assigned a unique identifier where jobs that have been run before could have detailed *profile* automatically assigned to their identifier by a combination of the *Allocator* and *CloudMonitor*. Then for future runs of these tasks the *Allocator* would have a detailed, automatic and true breakdown of the tasks characteristics.

The meta-data assigned to an instance request could allow the *Allocator* to automatically choose the most appropriate VM size and configuration for the job without the user having to specify which flavour of VM they would like. Making this decision automatically could eliminate waste and has potential to optimize the system further. In such a situation, for example, a larger or faster VM could be assigned to a user if there is capacity to do so and it does not excessively impact energy usage. This may even lead to a situation where the idea of virtualization of a machine is abstracted away from the end user and they simply submit the task they want completed along with meta-data governing how it should be run.

8 References

- [1] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (draft). NIST special publication, 800(145), 7.
- [2] Creeger, M. (2009). CTO roundtable: cloud computing. *Commun. ACM*, 52(8), 50-56.
- [3] Koomey, J. (2011). Growth in data center electricity use 2005 to 2010. Oakland, CA: Analytics Press. August, 1, 2010.
- [4] National Computing Centre Limited & National Computing Centre Limited Staff (2008). *The Green IT Paradox: Results of the NCC Rapid Survey*. John Wiley & Sons, Incorporated
- [5] Fan, X., Weber, W. D., & Barroso, L. A. (2007). Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News*, 35(2), 13-23.
- [6] Bohra, A. E., & Chaudhary, V. (2010, April). VMeter: Power modelling for virtualized clouds. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* (pp. 1-8). Ieee.
- [7] Chen, Q., Grosso, P., van der Veldt, K., de Laat, C., Hofman, R., & Bal, H. (2011, December). Profiling energy consumption of VMs for green cloud computing. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on* (pp. 768-775). IEEE.
- [8] Chen, F., Schneider, J., Yang, Y., Grundy, J., & He, Q. (2012, June). An energy consumption model and analysis tool for cloud computing environments. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on* (pp. 45-50). IEEE.
- [9] Kansal, A., Zhao, F., Liu, J., Kothari, N., & Bhattacharya, A. A. (2010, June). Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on cloud computing* (pp. 39-50). ACM.
- [10] Srikantaiah, S., Kansal, A., & Zhao, F. (2008, December). Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems* (Vol. 10). USENIX Association.
- [11] Chen, G., He, W., Liu, J., Nath, S., Rigas, L., Xiao, L., & Zhao, F. (2008, April). Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *NSDI* (Vol. 8, pp. 337-350).
- [12] Kim, H., Lim, H., Jeong, J., Jo, H., & Lee, J. (2009, March). Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS*

international conference on Virtual execution environments(pp. 101-110). ACM.

- [13] Moreno, I. S., Yang, R., Xu, J., & Wo, T. (2013, March). Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement. In *Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on* (pp. 1-8). IEEE.
- [14] Liao, J. S., Chang, C. C., Hsu, Y. L., Zhang, X. W., Lai, K. C., & Hsu, C. H. (2012, September). Energy-Efficient Resource Provisioning with SLA Consideration on Cloud Computing. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on* (pp. 206-211). IEEE.
- [15] Corradi, A., Fanelli, M., & Foschini, L. (2012). VM consolidation: a real case based on OpenStack Cloud. *Future Generation Computer Systems*.
- [16] Beloglazov, A., & Buyya, R. (2012). OpenStack neat: A framework for dynamic consolidation of virtual machines in OpenStack clouds–A blueprint. Technical Report CLOUDS-TR-2012-4, cloud computing and Distributed Systems Laboratory, The University of Melbourne.
- [17] Lindgren, H. (2013). Performance Management for Cloud Services: Implementation and Evaluation of Schedulers for OpenStack.
- [18] Webb, M. (2008). Smart 2020: Enabling the low carbon economy in the information age. The Climate Group. London, 1(1), 1-1.
- [19] Yamini, R. (2012, March). Power management in cloud computing using green algorithm. In *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on* (pp. 128-133). IEEE.
- [20] Beloglazov, A., Buyya, R., Lee, Y. C., & Zomaya, A. (2011). A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2), 47-111.
- [21] Brown, D. J., & Reams, C. (2010). Toward energy-efficient computing. *Communications of the ACM*, 53(3), 50-58.
- [22] Barroso, L. A., & Holzle, U. (2007). The case for energy-proportional computing. *Computer*, 40(12), 33-37.
- [23] Shankland, S. (2005). Power could cost more than servers, Google warns. CNET news.com, URL <http://news.com.com/2100-1010,3-5988090>
- [24] Wong, K. (Ed.). (2013). Change Brewing in the Power Game. *Desktop Engineering*, 18(10), 33-36.

- [25] Intel Press Release (June 4th 2013): 4th Generation Intel® Core™ Ushers New Wave of 2-in-1 Devices
http://newsroom.intel.com/community/intel_newsroom/blog/2013/06/03/4th-generation-intel-core-ushers-new-wave-of-2-in-1-devices
- [26] Wong, G. (2013). SSD Market Overview. In *Inside Solid State Drives (SSDs)*(pp. 1-17). Springer Netherlands.
- [27] Mayo, R. N., & Ranganathan, P. (2005). Energy consumption in mobile devices: why future systems need requirements-aware energy scale-down. In *Power-Aware Computer Systems* (pp. 26-40). Springer Berlin Heidelberg.
- [28] Greenberg, S., Mills, E., Tschudi, B., Rumsey, P., & Myatt, B. (2006). Best practices for data centers: lessons learned from benchmarking 22 data centers. *Proceedings of the ACEEE Summer Study on Energy Efficiency in Buildings in Asilomar, CA*. ACEEE, August, 3, 76-87.
- [29] Schien, D., Shabajee, P., Wood, S. G., & Preist, C. (2013, May). A model for green design of online news media services. In *Proceedings of the 22nd international conference on World Wide Web* (pp. 1111-1122). International World Wide Web Conferences Steering Committee.
- [30] Vodafone McLaren Mercedes press release (5th December 2011). Vodafone McLaren Mercedes becomes the world's first ever carbon-neutral Formula 1 team. <http://www.mclaren.com/formula1/team/vodafone-mclaren-mercedes-becomes-the-worlds-first-ever-carbon-neutral-formula-1-team/>
- [31] Nichols, W. (June 7th 2013): McLaren races into green data centre development. *Business Green*. <http://www.businessgreen.com/bg/news/2273336/mclaren-races-into-green-data-centre-development>
- [32] Yu, Y., Bhatti S. Building Energy Awareness into ICT Systems: A lab computer user energy awareness and usage experiment. (January 2012). http://mist.cs-st-andrews.ac.uk/wp-uploads/2012/01/year2_poster.jpg
- [33] Liikkanen, L. (2009). Extreme-user approach and the design of energy feedback systems. In *International Conference on Energy Efficiency in Domestic Appliances and Lighting* (pp. 16-18).
- [34] Pelley, S., Meisner, D., Wenisch, T. F., & VanGilder, J. W. (2009, June). Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*.
- [35] S. Chen, Y. Hu, and L. Peng, "Optimization of Electricity and Server Maintenance Costs in Hybrid Cooling Data Centers," In *Proceedings of The 6th IEEE International Conference on cloud computing (CLOUD)*, Santa Clara, CA, Jun. 2013.

- [36] Brown, R. (2008). Report to congress on server and data center energy efficiency: Public law 109-431.
- [37] Global e-Sustainability Initiative. (2008). SMART 2020: Enabling the low carbon economy in the information age. press release, Brussels, Belgium, June, 20.
- [38] Forrest, W., Kaplan, J. M., & Kindler, N. (2008). Data centers: how to cut carbon emissions and costs. McKinsey on business technology, 14(6).
- [39] Accenture PLC. cloud computing and Sustainability: The environmental Benefit of Moving to the Cloud. Report, 2010. Available at: http://www.accenture.com/SiteCollectionDocuments/PDF/Accenture_Sustainability_Cloud_Computing_TheEnvironmentalBenefitsofMovingtotheCloud.pdf
- [40] Garg, S. K., & Buyya, R. (2020). Green cloud computing and environmental sustainability. *Harnessing Green IT: Principles and Practices*, 315-340.
- [41] Meisner, D., Gold, B. T., & Wenisch, T. F. (2009, March). PowerNap: eliminating server idle power. In *ACM Sigplan Notices* (Vol. 44, No. 3, pp. 205-216). ACM.
- [42] Pinheiro, E., Bianchini, R., Carrera, E. V., & Heath, T. (2001, September). Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on compilers and operating systems for low power* (Vol. 180, pp. 182-195).
- [43] Felter, W., Rajamani, K., Keller, T., & Rusu, C. (2005, June). A performance-conserving approach for reducing peak energy consumption in server systems. In *Proceedings of the 19th annual international conference on Supercomputing*(pp. 293-302). ACM.
- [44] S. Zanikolas and R. Sakellariou, "A taxonomy of grid monitoring systems" in *Future Generation Computer Systems*, vol. 21, no. 1, pp. 163-188, 2005.
- [45] M. Massie, *et al.*, "The ganglia distributed monitoring system: design, implementation, and experience" in *Parallel Computing*, vol. 30, no. 7, pp. 817-840, 2004.
- [46] Zhang, X., Freschl, J. L., & Schopf, J. M. (2003, June). A performance study of monitoring and information services for distributed systems. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (pp. 270-281). IEEE.
- [47] Czajkowski, K., Fitzgerald, S., Foster, I., & Kesselman, C. (2001). Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on* (pp. 181-194). IEEE.
- [48] "DataGrid Information and Monitoring Services Architecture: Design, Requirements and Evaluation Criteria", Technical Report, DataGrid, 2002.

- [49] Livny, M., & Solomon, M. (2009). Condor Hawkeye Project. Univ. of Wisconsin-Madison, <http://www.cs.wisc.edu/condor/hawkeye>.
- [50] Chung, W. C., & Chang, R. S. (2009). A new mechanism for resource monitoring in Grid computing. *Future Generation Computer Systems*, 25(1), 1-7.
- [51] Lucas Jr, H. (1971). Performance evaluation and monitoring. *ACM Computing Surveys (CSUR)*, 3(3), 79-91.
- [52] Subramanyan, R., Miguel-Alonso, J., & Fortes, J. A. (2000, November). A scalable SNMP-based distributed monitoring system for heterogeneous network computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (p. 14). IEEE Computer Society.
- [53] A simple network management protocol (SNMP). Network Information Center, SRI International, 1989.
- [54] Tierney, B., Johnston, W., Crowley, B., Hoo, G., Brooks, C., & Gunter, D. (1998, July). The NetLogger methodology for high performance distributed systems performance analysis. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on* (pp. 260-267). IEEE.
- [55] Molka, D., Hackenberg, D., Schone, R., & Muller, M. S. (2010, August). Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors. In *Green Computing Conference, 2010 International* (pp. 123-133). IEEE.
- [56] Dargie, W., & Schill, A. (2012, June). Analysis of the Power and Hardware Resource Consumption of Servers under Different Load Balancing Policies. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (pp. 772-778). IEEE.
- [57] Pelley, S., Meisner, D., Zandevakili, P., Wenisch, T. F., & Underwood, J. (2010, March). Power routing: dynamic power provisioning in the data center. In *ACM Sigplan Notices* (Vol. 45, No. 3, pp. 231-242). ACM.
- [58] Bohrer, P., Elnozahy, E. N., Keller, T., Kistler, M., Lefurgy, C., McDowell, C., & Rajamony, R. (2002). The case for power management in web servers. In *Power aware computing* (pp. 261-289). Springer US.
- [59] Elnozahy, E. M., Kistler, M., & Rajamony, R. (2003). Energy-efficient server clusters. In *Power-Aware Computer Systems* (pp. 179-197). Springer Berlin Heidelberg.
- [60] Economou, D., Rivoire, S., Kozyrakis, C., & Ranganathan, P. (2006, June). Full-system power analysis and modeling for server environments. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulation* (pp. 70-77).

- [61] Stoess, J., Lang, C., & Bellosa, F. (2007, June). Energy Management for Hypervisor-Based virtual machines. In USENIX annual technical conference(pp. 1-14).
- [62] Bertran, R., Becerra, Y., Carrera, D., Beltran, V., Gonzalez, M., Martorell, X., ... & Ayguade, E. (2010, October). Accurate energy accounting for shared virtualized environments using PMC-based power modeling techniques. In Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on (pp. 1-8). IEEE.
- [63] Hewlett-Packard, Microsoft, Phoenix, and Toshiba (2004). Advanced configuration and power interface specification.
- [64] Yu, Y., & Bhatti, S. (2010, September). Energy measurement for the cloud. In Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on (pp. 619-624). IEEE.
- [65] Priya, B., Pilli, E. S., & Joshi, R. C. (2013, February). A survey on energy and energy consumption models for Greener Cloud. In Advance Computing Conference (IACC), 2013 IEEE 3rd International (pp. 76-82). IEEE.
- [66] Brebner, P., O'Brien, L., & Gray, J. (2009, September). Performance modelling energy consumption and carbon emissions for Server Virtualization of Service Oriented Architectures (SOAs). In Enterprise Distributed Object Computing Conference Workshops, 2009. EDOCW 2009. 13th (pp. 92-99). IEEE.
- [67] Chaudhary, V., Cha, M., Walters, J. P., Guercio, S., & Gallo, S. (2008, March). A comparison of virtualization technologies for HPC. In Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on (pp. 861-868). IEEE.
- [68] McDougall, R., & Anderson, J. (2010). Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4), 40-56.
- [69] El Rheddane, A., De Palma, N., Boyer, F., Dumont, F., Menaud, J. M., Tchana, A., ... & Lina—Nantes, F. (2013, July). Dynamic Scalability of a Consolidation Service. In International Conference on cloud computing (pp. 01-09).
- [70] Ahmad, F., & Vijaykumar, T. N. (2010, March). Joint optimization of idle and cooling power in data centers while maintaining response time. In *ACM Sigplan Notices* (Vol. 45, No. 3, pp. 243-256). ACM.
- [71] Qouneh, A., Li, C., & Li, T. (2011, November). A quantitative analysis of cooling power in container-based data centers. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on* (pp. 61-71). IEEE.
- [72] Petrucci, V., Loques, O., & Mossé, D. (2009, May). A dynamic configuration model for

power-efficient virtualized server clusters. In 11th Brazilian Workshop on Real-Time and Embedded Systems (WTR) (Vol. 2).

- [73] Voorsluys, W., Broberg, J., Venugopal, S., & Buyya, R. (2009). Cost of virtual machine live migration in clouds: A performance evaluation. In *cloud computing* (pp. 254-265). Springer Berlin Heidelberg.
- [74] Nathuji, R., Kansal, A., & Ghaffarkhah, A. (2010, April). Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems* (pp. 237-250). ACM.
- [75] Pinheiro, E., Weber, W. D., & Barroso, L. A. (2007, February). Failure Trends in a Large Disk Drive Population. In *FAST* (Vol. 7, pp. 17-23).
- [76] Chun, B. G., Iannaccone, G., Iannaccone, G., Katz, R., Lee, G., & Niccolini, L. (2010). An energy case for hybrid datacenters. *ACM SIGOPS Operating Systems Review*, 44(1), 76-80.
- [77] Mars, J., Tang, L., Hundt, R., Skadron, K., & Soffa, M. L. (2011, December). Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 248-259). ACM.
- [78] Wuhib, F., Stadler, R., & Lindgren, H. (2012, October). Dynamic resource allocation with management objectives—Implementation for an OpenStack cloud. In *Network and Service Management (CNSM), 2012 8th International Conference on* (pp. 309-315). IEEE.
- [79] Graubner, P., Schmidt, M., & Freisleben, B. (2011, July). Energy-efficient management of virtual machines in eucalyptus. In *cloud computing (CLOUD), 2011 IEEE International Conference on* (pp. 243-250). IEEE.
- [80] Lazar, J., Feng, J. H., & Hochheiser, H. (2010). Research methods in human-computer interaction. Wiley. com. Chapter 3 “Experimental Research” pg 19-40
- [81] Nunamaker Jr, J. F., & Chen, M. (1990, January). Systems development in information systems research. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on* (Vol. 3, pp. 631-640). IEEE.
- [82] Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, 28(1), 75-105.
- [83] Feitelson, D. G. (2006). Experimental computer science: The need for a cultural change.
- [84] Rosenthal, R & Rosnow, R. (1991). *Essentials of behavioural research*. McGraw-Hill.
- [85] Cornford, T., & Smithson, S. (2006). *Project research in information systems: a student's*

guide. Palgrave. Chapter 8 "Analysing Research Data" pg 127-155.

- [86] McDougall, R., & Anderson, J. (2010). Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4), 40-56.
- [87] Tang, W., Fu, Y., Cherkasova, L., & Vahdat, A. (2003, June). Medisyn: A synthetic *streaming* media service workload generator. In *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video* (pp. 12-21). ACM.
- [88] Litzkow, M. J., Livny, M., & Mutka, M. W. (1988, June). Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on* (pp. 104-111). IEEE.
- [89] Macdonald, Angus. PhD Thesis, "An Autonomic, Resource-Aware, Workstation-based Distributed Database System", University of St Andrews, 2012
- [90] Kirby, G., *et al.*, "NUMONIC", 2010 [Computer Program, available via <http://blogs.cs.st-andrews.ac.uk/numonic/>, 03-Mar-2011].
- [91] Khajeh-Hosseini, A., Greenwood, D., Smith, J. W., & Sommerville, I. (2012). The Cloud Adoption Toolkit: supporting Cloud adoption decisions in the enterprise. *Software: Practice and Experience*, 42(4), 447-465.
- [92] Larabel, M., & Tippett, M. (2011). *Phoronix* test suite. <http://www.phoronix-test-suite.com/>
- [94] OpenStack cloud computing project, <http://www.OpenStack.org/>
- [94] Lutz, M. (2013). *Learning python*. O'Reilly Media.

Appendix A – *Allocator* User Guide

To interact with the energy-efficient *Allocator* software for OpenStack as an end user, there is only one additional command that is required.

When requesting a new virtual machine instance from nova, add the flag “—hint vm_type=<type>” to the end of your boot command. For example:

```
nova boot test --image "my-image" --flavor 3
```

becomes:

```
nova boot test --image "my-image" --flavor 3 --hint vm_type=CPU
```

```
cloud@cloud2-node1-46:~$ nova boot test --image "ubuntu" --flavor 3 --key_name mykey --hint vm_type=CPU
+-----+-----+
| Property | Value |
+-----+-----+
| OS-DCF:diskConfig | MANUAL |
| OS-EXT-SRV-ATTR:host | None |
| OS-EXT-SRV-ATTR:hypervisor_hostname | None |
| OS-EXT-SRV-ATTR:instance_name | instance-00000693 |
| OS-EXT-STS:power_state | 0 |
| OS-EXT-STS:task_state | scheduling |
| OS-EXT-STS:vm_state | building |
| accessIPv4 | |
| accessIPv6 | |
| adminPass | imnaB5aGbURH |
| config_drive | |
| created | 2013-09-21T12:38:23Z |
| flavor | m1.medium |
| hostId | |
| id | b1707452-2613-4d45-8097-ff10f7d82f76 |
| image | ubuntu |
| key_name | mykey |
| metadata | {} |
| name | test |
| progress | 0 |
| status | BUILD |
| tenant_id | d997c41e6c3c46adb01330f1e987fb87 |
| updated | 2013-09-21T12:38:23Z |
| user_id | 4ecb9f4d84d04e7dbfcd85a261861897 |
+-----+-----+
cloud@cloud2-node1-46:~$
```

Figure 8.1: Example of using the Allocator software

The type of your VM can be one of: CPU, DISK, MEM or NET. Please assign the appropriate label based on the resource that you VM will consume most of. Providing accurate *vm_type* labels will result in increased application performance and reduced energy consumption. Failure to do so may result in degraded performance for your applications and higher than necessary system energy usage.

If you do not know the profile of your application, please run it once

alongside the *CloudMonitor* software³⁴ and examine the generated data to determine the correct label.

³⁴ <http://github.com/jws7/cloudmonitor>

Appendix B – *Allocator* Administrator Guide

To install the *Allocator* software for OpenStack follow these procedures:

1. Pull the latest version of the *Allocator* from OpenStack³⁵
2. Overwrite the local files with replacements from *Allocator*.
3. Set your nova.conf configuration file to:

```
# Filter:
--scheduler_available_filters=nova.scheduler.filters.standard_filters
--scheduler_default_filters=Allocator

# Weigher:
--least_cost_functions=nova.scheduler.least_cost allocator
--compute_fill_first_cost_fn_weight=-1.0
```

4. Set up DB access in the *sql/config.txt* file and execute the script *setup.sql*
5. Restart all *nova* services. The script included “*nova-restart.sh*” will do this for you.
6. For additional performance, make the following changes:

- a. Set the libvirt XML template on all hosts to have *writeback* cache mode
- b. Set the libvirt XML template on all hosts to use the “*io*” driver.
- c. Set disk elevators on the guest and host operating systems to *noop* and *deadline* respectively.
- d. Set the nova.conf on each host to include the lines:

```
--use_cow_images=false
--force_raw_images=true
```

³⁵ <http://github.com/jws7/allocator>

7. Modify the workload mix rules as you see fit by editing the *mix.rules* file.
8. To edit the code for your own purposes, first fork the repository on github before following steps 1-7.

Appendix C – *CloudMonitor* user guide

CloudMonitor

1. Download the latest code from github.com³⁶
2. Compile the code to a Java jar file, *CloudMonitor.jar*.
3. Setup the database access details in the file *sql/config.txt*
4. Run the database setup script *sql/setup.sql*
5. To run *CloudMonitor* execute:

```
java -jar CloudMonitor.jar &
```

EnergyMonitor

1. To run *EnergyMonitor* on a PDU, complete the same steps above for the *EnergyMonitor* class.
2. Include the PDU *MIB.txt* file in the same directory as the runnable jar.
3. Execute:

```
java -jar EnergyMonitor.jar <snmp socket> &
```

MemoryMonitor

1. Repeat the steps 1-5 for *MemoryMonitor* if required.

***PowerModel* generation**

1. Choose appropriate timestamps for the *training phase*
2. Input the machine ID and timestamps to the *GenerateModel* program and

³⁶ <http://github.com/jws7/cloudmonitor>

execute it. The model will then be stored under the chosen name for this machine configuration.

Appendix D - Publications

This thesis includes material from the following publications:

- Smith J W, Sommerville I. 2013. *Understanding tradeoffs between energy usage and Performance in a Virtualized Environment*. IEEE 6th Int. Conf. on Cloud Computing, Santa Clara, USA. 2013. DOI 10.1109/CLOUD.2013.138
- Smith J W, Khajeh-Hosseini A, Ward J S, Sommerville I. 2012. *CloudMonitor: Profiling Power Usage*. IEEE 5th Int. Conf. on Cloud Computing, Honolulu, USA. 2012. DOI 10.1109/CLOUD.2012.112
- Khajeh-Hosseini A, Greenwood D, Smith J W, Sommerville I. 2011. *The Cloud Adoption Toolkit: Supporting Cloud Adoption Decisions in the Enterprise*. Software: Practice and Experience. DOI: 10.1002/spe.1072