UiO : Department of Informatics
University of Oslo

Graph Caching in Near-Far Clouds using Emerald

An experimental analysis on Planetlab

Hichael Mareno Søbstad

Master's Thesis, Spring 2017

Graph Caching in Near-Far Clouds using Emerald

Hichael Mareno Søbstad

Spring 2017

Acknowledgements

It has been a great pleasure to work with Professor Eric Bartley Jul. I would argue that he has made me a better problem solver and big-picture thinker - and for that I wish to thank him. I would also like to thank my family for their continuous support.

Abstract

In modern distributed systems there exists two major bottlenecks that are holding back performance; dynamic memory and the speed of light. In this thesis the focus will be on the speed of light, and how one may reduce its impact on performance. For each remote invocation, there is an associated overhead that corresponds to the latency between the respective nodes. If latency is high and packet-exchange frequent, there is potential for *performance degradation.* The incorporation of our graph caching scheme for the near-far cloud architecture aims to reduce the the amount of remote invocations. It does so by taking advantage of key mechanisms provided by the distributed-objects programming language known as Emerald.

We have performed experiments on Planetlab, where two architectures has been compared; client-server and the near-far cloud architecture. Both architectures incorporated caching. A custom depth-first search was utilized in order to evaluate these architectures. Our findings indicate that caching with the near-far architecture may substantially reduce the total amount of invocations towards the remote node when compared to the client-server architecture. It also demonstrates how a reduction in total invocations may drastically improve performance. At the same token, if there exists frequent invocations to the remote node, and the latency is high - performance is to suffer.

Of course, there is more to the equation than simply latency. Local cache-lookups are potentially more expensive than that of latency. This is especially true at bigger cache sizes. If latency is too low, the overhead associated with frequent cache-look ups in bigger caches may potentially *overshadow* any performance gains made by the reduction in remote invocations. Thus, compromises have to be made to fully take advantage of the near-far cloud caching scheme.

These findings are restricted to our environment of study. We do not attempt to extrapolate results beyond this very scope.

CONTENTS

List of Figures **11**

1 Introduction **14**

1.1 Motivation 14

1.2 Problem Statement 15

1.2.1 The issue 15

1.2.2 The solution 15

1.3 Objectives 16

1.4 Approach 16

1.5 Work done 17

1.6 Evaluation 17

1.7 Results 17

1.8 Limitations 18

1.9 Conclusion 18

1.10 Future Work 18

1.11 Summary 19

2 Motivation **20**

2.1 Moore's law 20

2.2 Performance bottlenecks 22

2.2.1 The slow speed of DRAM 22

2.2.2 The slow speed of light 24

2.3 Summary 24

3 Background **25**

3.1 Distributed systems 26

3.1.1 Implications of distributed systems 26

3.1.2 Challenges in distributed systems 27

Heterogeneity 27

Openness 27

Scalability 28

Approaches to scaling 30

Improving scalability 30

Transparency 31

Access transparency 31

Location transparency 31

Concurrency transparency 31

Replication transparency 31

Failure transparency 32

Mobility transparency 32

Performance transparency 32

Scaling transparency 32

3.2 Memory hierarchy 33

3.2.1 Caching 35

Cache motivation 35

Multi-level cache 37

Key cache concepts 38

Cache performance 39

Cache layout strategies 40

Summary 41

3.3 Graph Theory 42

Graph Constituents 42

Graph types 42

An example graph 43

Depth-First Search 44

Complexity of DFS 44

Breadth-First Search 45

Complexity of BFS 45

3.4 Cloud computing 46

3.4.1 Cloud hierarchy 47

3.4.2 Near-far cloud 48

3.5 Emerald . . . 49
3.5.1 Object . . . 49
3.5.2 Object mobility . . . 49
3.5.3 Data Migration . . . 49
Locate . . . 50
Move . . . 50
Fix . . . 50
Unfix . . . 51
Refix . . . 51
3.5.4 Language Constituents . . . 52
Attached . . . 53
3.5.5 Emerald Type System . . . 55
Purpose of type system . . . 55
Type checking and error handling . . . 57
Transparency in Emerald . . . 58
3.6 Planetlab . . . 59
3.6.1 Step one . . . 59
3.6.2 Step two . . . 60
3.6.3 Step three . . . 61
3.7 Summary . . . 62

4 Problem statement **63**

4.1 The issue . . . 63

4.2 The solution 63

5 Design **65**

5.1 Scope 65

5.2 Architecture 66

5.2.1 Data Structure 66

5.2.2 Algorithms 66

5.3 Near-far cloud prototype 66

5.3.1 Caching process near-far cloud 68

5.4 Client-server prototype 70

5.4.1 Caching process client-server 70

5.5 Summary 71

6 Implementation **72**

6.1 Classes 73

6.1.1 Roles 74

Class Role 74

Class Client 75

Class Proxy 76

Class Data Center 77

6.1.2 Data structure 78

Class Cache 78

Typeobject CacheAlgorithm 78

Class DFSCaching 79

Class Graph 80
Class Vertex 81
Class edge 81
6.2 Key Algorithms 82
6.2.1 cacheData 82
6.2.2 expandVertex 83
6.2.3 partialExpansion 83
6.3 Summary 84

7 Evaluation **85**

7.1 Memory limitations in emerald 86
7.1.1 Implications of memory constraints 87
7.2 Performance metrics 88
7.2.1 Nodes and latency 88
7.2.2 Performance of local algorithms 89
Generate graph 90
Transmit graph 91
Recursive DFS 92
Proxy (Czech Republic) 93
Data Center (United States) 94
Comparison between nodes 95
New Data Center Node 95
Data Center (Canada) 96

7.2.3 Caching . . . 97
Expansive DFS . . . 97
Caching performance comparison . . . 98
Client-server caching (minimal latency) . . . 98
Client-server caching (high latency) . . . 99
Remote invocations . . . 100
Near-far cloud caching . . . 101
Partial caching . . . 102
Deep caching . . . 104
7.3 Analysis . . . 105
7.3.1 Findings . . . 105
7.3.2 Scalability . . . 106
7.4 Source Code . . . 106
7.5 Summary . . . 106

8 Limitations **108**

9 Conclusion **109**

10 Future Work **111**
10.1 Eviction Policy . . . 111
10.2 Smart Chained-proxies . . . 112
10.3 Parallel caching . . . 112
10.4 Summary . . . 112

11 Bibliography **113**

LIST OF FIGURES

2.1 The ever growing gap between memory and CPU performance [16]. . 23

3.1 Illustration of a traditional memory hierarchy [8]] 33

3.2 Illustration of a two-level cache [10] 37

3.3 Illustration of our example graph [24] 43

3.4 Recursive depth-first search [24] 44

3.5 Breadth-First Search [24] 45

3.6 Illustration of the cloud hierarchy [24] 47

3.7 Illustration of the near-far cloud [24] 48

3.8 "Kilroy", a classic emerald program 52

3.9 Class Pointer 53

3.10 Class List 54

5.1 Illustration of the near-far cloud [24] 67

5.2 Near-far caching scheme [24] 68

5.3 Client-server caching scheme [24] 70

6.1 Class diagram outlining the components of the system 73

6.2 Functionality required in order to conform to type Role 74

6.3 Attributes and functionality associated with Client 75

6.4 Attributes and functionality associated with Proxy 76

6.5 Attributes and functionality associated with DataCenter 77

6.6 Functionality required in order to conform to type CachingAlgorithm 78

6.7 Attributes and functionality associated with DFSCaching 79

6.8 Attributes and functionality associated with Graph 80

6.9 Attributes and functionality associated with Vertex 81

6.10 Attributes and functionality associated with Edge 81

6.11 cacheData 82

6.12 expandVertex 83

6.13 expandVertex 83

7.1 Out of memory prompt 86

7.2 Memory runs out at approximately 652000 bytes 86

7.3 Time taken to generate graph 87

7.4 Illustration of the Planetlab nodes [24] 88

7.5 Ping results towards data center 88

7.6 Ping results towards proxy 89

7.7 Graph generation performance 90

7.8 Graph transmission overheads 91

7.9 DFS worst-case at size 1000 93

7.10 DFS worst-case at size 6000 93

7.11 DFS worst-case at size 1000 94

7.12 DFS worst-case at size 6000 94

7.13 Latency measures 95

7.14 DFS worst-case at size 6000 96

7.15 Client-server caching with server of minimal latency 98

7.16 Client-server caching with server of high latency 99

7.17 Invocations associated with client-server caching 100

7.18 Near-far partial caching [50|2500|5000] 102

7.19 Near-far partial caching [10|1000|5000] 103

7.20 Near-far deep caching [25|250|5000] 104

7.21 Near-far deep caching [25|250|5000] 105

CHAPTER 1

INTRODUCTION

The aim of this chapter is to provide the reader with a general understanding of our thesis. It is divided into sections that corresponds to the chapters of the thesis, which are structured in chronological order. Each section provides a brief summary of the given chapter. This will ease further reading by providing a wider context.

1.1 Motivation

There exists two main factors that are restricting performance in distributed systems. The *slow* speed of DRAM and the *slow* speed of light holds back the performance of the *incredible fast* CPU. Our main focus will be on latency, because of its as its importance in distributed systems. It is also a *true* bottleneck, since the theoretical speed of light *is not subject to change.* New memory technologies may be invented as better alternatives to DRAM, but the speed of light will remain.

1.2 Problem Statement

1.2.1 The issue

In distributed systems where latency is high and packet exchange frequent, there is potential for performance degradation. Since one cannot alter the latency that exists between two nodes, our only option is to *reduce the frequency* of invocations. By reducing the amount of invocations towards remote nodes, one may potentially improve performance.

1.2.2 The solution

This can best be done by the act of caching. Before any remote invocation, one is first to consult the local cache. If the desired data is identified, there is no need for a remote invocation. However, if the data is not identified, a remote invocation will be performed. Whenever a cache hit occurs at the remote node, the data will be transmitted to the client and integrated with its cache. This way, there is an increasing probability that future accesses will be towards the local cache rather than the cache of the remote node.

The concept of caching may be further expanded upon with the incorporation of the near-far cloud concept. The near-far cloud concept revolves around the idea that one may take advantage of the physical location of a given cloud service. The location of a service or server comes with both advantages and disadvantages. Services that are located far from the client (at the far-cloud) provides good computational power and exceptional storage capacity. However, its disadvantage is that it exhibits high latency towards the client. Services that are located near the client (in the near-cloud) provides both acceptable computational power and decent storage capabilities. Seen from a hardware perspective, it is inferior to that of the service located far into the cloud, but there is a catch. The major advantage of having services located at the near-cloud is that these will exhibit minimal latency to that of the client.

In a classic client-server architecture, the client is performing multiple, potentially expensive invocations towards a data center, which is located far away and as

such exhibits high latency towards the client. Even if caching is incorporated, there will still be excessive remote invocations that may potentially degrade performance. Instead, an intermediary node, referred to as proxy is deployed to do the client's bidding. The intermediary node is located physically close to the client, and as such exhibit minuscule latency towards it. This will offload the network of the client.

Both the proxy and the client are equipped with caching capabilities. Whenever a cache miss occurs on the proxy node, the data center will be consulted and the desired piece of data fetched. The data will then be integrated with the cache of the proxy. The nodes form a multi-level caching hierarchy. In an event of a cache miss at the client side, one is to perform remote invocations towards the proxy node. These are considerable cheaper than remote invocations towards the data center, because of the difference in latency. Again, if the desired data is not present on the proxy node, the proxy will perform a remote invocation to the data center on behalf of the client. The resulting data will then get integrated at both the proxy and the client. We are using a graph data structure to implement our caches. This is done to take full advantage of its implementation language, Emerald.

1.3 Objectives

The goal of our thesis is to contribute to the existing pool of research on the near-far cloud concept as well as the Emerald programming language, which has proven essential to our study.

1.4 Approach

The thesis is of experimental character. We want to identify whether our proposed near-far caching solution may improve performance by reducing remote invocations to the data center. This is rooted in our literature study, where latency has been highlighted as a major bottleneck of performance. Two prototypes have been implemented in the distributed-objects language Emerald. We have then conducted several experiments on the global testbed for distributed applications, known as Planetlab. The results from our experiments have been subject to analysis, and conclusions drawn.

1.5 Work done

We have developed two prototypes. One that conforms to a classic client-server architecture, and one that is based on the near-far cloud concept. Both are incorporating graph caching. They have both been developed in Emerald and evaluated on Planetlab.

1.6 Evaluation

Several experiments have been conducted on Planetlab. We have benchmarked performance of local cache look-ups, where the relationship between cache size and look-up speed have been investigated. Experiments to establish the relationship between data size and transmission time for remote invocations have also been carried out. Finally, we have compared our two caching prototypes to one another and analyzed the results.

1.7 Results

The results indicate that performance degradation do indeed occur when latency is high and packet-exchange frequent. Performance associated with our client-server prototype heavily depends on the latency towards the server. There is an almost *exponential* increase in time spent caching when comparing a server of *minuscule* latency to one of *high* latency. We have ascertained through experiments that the high latency server is equipped with better computational power than that of the minuscule latency server. Therefore, it is of our opinion that the observed discrepancy cannot be rooted in a difference of computational power.

Our near-far caching prototype results in substantially less invocations to the data center when compared to our client-server prototype. As for performance, there are two factors that influence this. As the cache grows in size, the overhead associated with cache look-ups increases. At bigger cache sizes, the overhead is *significant*. Our findings indicate that the cache look-look-up speed associated with bigger caches is so slow that it may potentially alleviate any gains made by reducing

remote invocations.

For our caching scheme to be effective, one has to either operate with smaller to medium sized caches or to perform remote calls to nodes of higher latency. We cover an extension of the near-far cloud architecture in our future work chapter.

1.8 Limitations

Our prototype was developed in Emerald and evaluated on Planetlab. Therefore, the associated results are limited to this environment. Our experiments performed on Planetlab are restricted to the exact nodes that we utilized. We do not attempt to generalize our findings to other environments.

1.9 Conclusion

It is our conclusion that the near-far graph caching scheme provides a significant reduction in remote invocations to the data center in comparison to the client-server graph caching. Performance gains depends on cache size and latency towards the data center. At small to medium sized caches with high latency data centres, the scheme provides a substantial performance increase in comparison to the client-server scheme. However, at bigger caches and low latency data centers the scheme suffers. This is due to the overhead associated with cache look-ups being too big in comparison to the performance gains associated with a reduction in remote invocations.

1.10 Future Work

Based on our results, we have come up with an extension of the near-far graph caching concept. In our extension, there are multiple levels of proxies that form a chain of inclusive caches towards the the data center. The first levels are of small size, guaranteeing quick cache look-ups. All caches are equipped with LRU (least recently used) eviction policy. Cache look-ups may be performed in parallel to speed up the caching process.

1.11 Summary

We have covered the core constituents that make up this thesis. Topic, scope, goal, results and the degree to which we are able to extrapolate these results have been covered. The reader is to have a fundamental understanding of what we wish to convey with our thesis, which in turn will ease further reading.

CHAPTER 2

MOTIVATION

In this section we are to present the underlying issues that sparked our interest in the topic of graph-caching in near-far clouds. We are to discuss Moore's law, and why the ever growing gap between CPU clock rate and memory access speed is of concern. The slow speed of light will also be covered, which is the key limiting factor of performance in distributed systems where latency is of concern. We are not to dwell into what can be done to alleviate these concerns, as this is covered in our problem statement chapter.

2.1 Moore's law

Moore's law is an observation made in 1965 by the co-founder of Intel, Gordon E. Moore. The original observation was that the number of transistors per square inch on integrated circuits doubles every year [19], however; this was revised to every two years in 1975 [23]. Recently Intel stated that the doubling occurs every 2.5 years [25]. Moore and other experts in the field prophesies that the observation will hold for at least two more decades. It should be noted that even though it has been coined a "law", it should be referred to more as a prediction; which it originally was. Some claim that this prediction is more of a self-fulfilling prophecy to drive the industry [23], and as a matter of fact, Gordon himself states this in 1996 when discussing his prediction: "More than anything, once something like this gets established, it

becomes more or less a self-fulfilling prophecy. The Semiconductor Industry Association puts out a technology road map, which continues this generation [turnover] every three years. Everyone in the industry recognizes that if you don't stay on essentially that curve they will fall behind. So it sort of drives itself" [23].

In recent years the progression has been gradually slowing down, and it is predicted by Gordon himself to halt completely sometime in the near future, unless there are major innovations being done. There are two main issues that poses a threat to the survival of Moore's Law. The first cause of issue is the one that is currently slowing down the progress, namely that of power consumption [17]. There is simply too many transistors stacked on single chip for the current materials to withstand the heat generated. The other issue is something that is not currently in effect, but may severely slow it down in the future. It is referred to as "The 5 nanometer problem". The 5nm problem refers to the size of the transistor nodes. When transistor nodes reaches the critical point of 5nm, an effect occurs. The effect is called quantum tunneling [18], and great measures have to be taken in order to deal with it.

2.2 Performance bottlenecks

2.2.1 The slow speed of DRAM

Moore's law was merely intended to be a predictive theory that referred to an increase in number of transistors per square inch on integrated circuits. For the first few years, it did contribute to substantial performance gains in computers, and more importantly; the advent of the microcomputer. However, as the years passed on, the overall performance gains on a system derived from a more powerful CPU started to slow down.

The reason for this is that the clock frequency associated with a processor is not the sole determinant of performance. The processor on-chip caches, the hard drive, the RAM and even the motherboard will influence the overall performance of a given computer. When the CPU is to decode and execute a set of instructions, it first needs to fetch the actual instructions. This implies having to consult the RAM; which then again has to consult the Hard Disk (issuing a page fault) if it isn't available. The CPU may have the clock frequency of a thousand dreams, yet it's performance will be severely restricted if the RAM can't keep up [5]. It has been stated that in recent years the RAM has become the bottleneck in Moore's Law [5]. As can be seen from figure 2.1, there exists an ever increasing discrepancy between that of the access speed of the CPU compared to RAM.

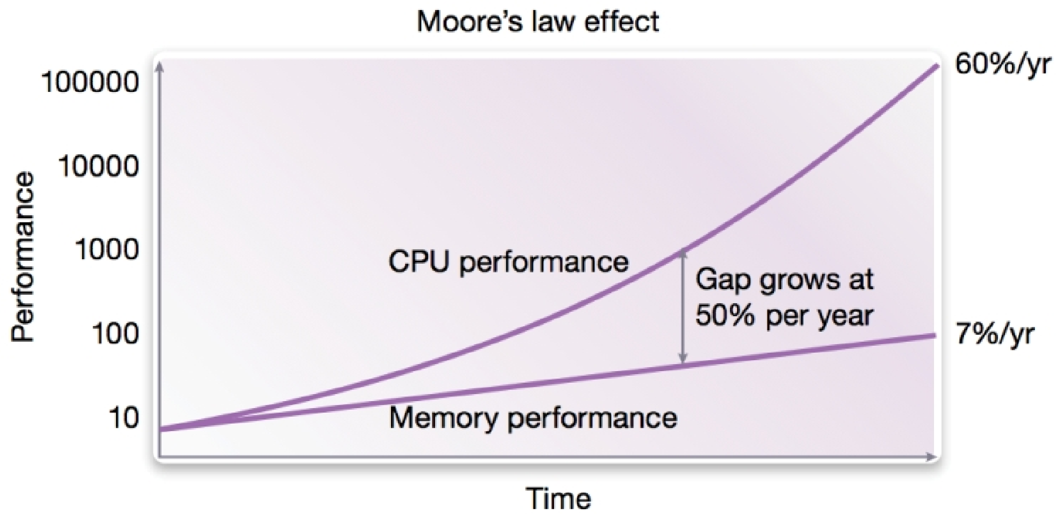


Figure 2.1: The ever growing gap between memory and CPU performance [16].

There are new technologies that are being developed as we speak to address the slow speed of memory, where the most promising one is spintronics. Sprintronics is a potentially revolutionary type of memory where magnetism is used instead of electrons to manipulate bits. It has the potential to be substantially faster than the modern DRAM technology [22].

However, even if spintronics do succeed in creating the next generation of memory, it is realistic to assume that standard RAM will still be around for some time.

2.2.2 The slow speed of light

In the world of distributed systems, there exists yet another performance bottleneck that has more profound implications on performance; the slow *speed of light*. The theoretical speed of light is 299 792 458 m/s [21]. Given the curvature of fiber-optics cables, one should expect this value to be roughly 2/3 of this. *Are we really saying that 199 861 638 m/s is slow?*.

When it comes to networked computing, there are other factors to consider as well; several interactions between components exists. The speed of light has be compared to that of the clock cycle rate of the CPU, which by today's standards are *incredible fast*. The fact of the matter is that the CPU stalls several hundred cycles waiting for data to arrive across the network. It is deemed a bottleneck because of the fact that the speed of light is an universal constant that can not be changed.

In distributed systems where nodes are located far from one another, and packet exchange is frequent, performance will suffer.

2.3 Summary

In this chapter we have covered two factors that restricts performance in modern distributed systems. The slow access speed of dynamic memory and the slow speed of light are deemed bottlenecks. A consequence of the speed of light is that frequent invocations to high latency nodes are expensive. One cannot improve the speed of light as it is a universal constant.

CHAPTER 3

BACKGROUND

This chapter aims to provide the reader with the theoretical foundations of our thesis. It will cover the following topics in chronological order: distributed systems, memory technology, graph theory, the Emerald programming language and Planetlab.

3.1 Distributed systems

A distributed system is according to Coulouris et al defined as "A system in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages". [7]

3.1.1 Implications of distributed systems

There are three main implications of a distributed system that is being discussed by Coulouris et al; namely that of *concurrency*, *lack of global clock* and *independent failures*. [7]

In a network of computers, concurrency is necessary to foster collaborate work. Coordination of resource-sharing programs executing concurrently is an important topic in distributed systems.

Considering that the only means of communication in a distributed system is by message passing, *associated times will change rapidly across nodes.* This is especially true for distributed systems encompassing entire continents, as the introduction of new time-zones would complicate matters even more. This **lack of global clock** leads to the facts the the application developer needs to consider how packets being sent are ordered not merely in time, but also in space. To achieve a realistic view of the current time, one should utilize logical clocks or vector clocks; where the latter is the most accurate choice.

The world of distributed systems is volatile. At one point in time a node is up and everything works perfect, the next split-second it is down. **Independent failures** refer to the fact that some components of the system may fail while others are still up and running. The node that crashed may or may not be discovered, it may or may not elicit hostile behaviour that can potentially invalidate the current state of the network. Thus, the lifespan of a node in a distributed system is *unpredictable*.

3.1.2 Challenges in distributed systems

In the Coulouris et al textbook "Distributed Systems: Concepts and Design" [7], there are *eight key challenges* being mentioned. We will provide a brief description of those that are relevant to thesis: heterogeneity, openness, scalability and transparency.

Heterogeneity

The Internet is composed of systems, programs and components that differs in architecture, operating systems and means of operation. These characteristics of heterogeneity pose challenges for distributed systems, because one system may be tailored for a specific architecture, and does not take into account other architectures that differs from that of the established one. The solution to this problem is Middleware technology. Middleware is a software components that can be seen as an added layer to an existing system. Its main purpose is to provide abstraction by masking heterogeneity through establishing a common interface through which all communication proceeds. The interface will marshal and demarshal packages so they all fit the protocol. CORBA is an example of a distributed-component that is being used as a middleware for heterogeneous environments, where programming languages and architectures may differ.

Openness

Openness is to which extent a system can be extended with software modules, or to modify the currently existing modules. In distributed systems, the term is specifically used in conjunction with the implementation of new resource-sharing services where the client programs can utilize their new service. The term published is used when talking about interfaces in open systems. A published interface is extensible and modifiable. An open distributed system employs a uniform communication mechanism in conjunction with published interfaces for access to shared resources.

Scalability

According to AT&T's Andre Bondi, scalability is the capability of a system, network or process to handle increasing amount of work and it's ability to be enhanced in order to cope with this growth [4]. If the cost of enlarging a system is too great, it is said to be unscalable. The cost may be unsatisfactory response times or memory requirements that surpasses that of available memory, which inevitably will lead to a crash. These factors may ultimately lead to customers leaving your service. Performance of a system should be measured not only in space and time complexity, but also its ability in its ability to scale gracefully. It is argued that scalability may be even more important than space and time complexity, as modern day applications and services are increasingly dependant upon users en-masse.

In Bondi's paper "Characteristics of Scalability and Their Impact on Performance" he described four types of scalability: load scalability, space scalability, space-time scalability and structural scalability.

Load scalability is the ability of a system to utilize the available resources while exhibiting stable performance. Its performance should remain unaffected by the degree of load being experienced. One may improve load scalability by adequately scheduling resources, avoid access algorithms that lead to self-expansion and to exploit parallelism.

Algorithms for fetching data from servers should be decentralized so that one does not fetch from the same source database; something that could lead to performance impairment if multiple users are accessing this single source. In order to load-scale systems effectively, one has to make sure that the various locations where data are stored are spread out. The Domain Name Server (DNS) used to have a single master file that would get distributed out to users on request. This became an inherent bottleneck of the system as the amount of users grew out of proportions compared to the predicament of the developers.

The increased demand for a specific service will eventually lead to a situation in which the current available resources that support the service is insufficient. The hardware supporting the service has to be extended in order to further scale the system. The physical location of the hardware has to be separate in order to handle increased access on the user-end and failures associated with the crashing of nodes.

The quantity of physical resources required to support a scalable system of N amount of users should never surpass that of O(N).

Space scalability is related to the memory capacity of a given system. A system is regarded as space-scalable if the required memory of a given service does not surpass that of the available memory of the system. The choice of data structures heavily influences space scalability. A data structure is categorized as being space-scalable if its memory requirements increases at most sublinearly with the number of elements. Compression might be used to reduce size of elements, however, this may conflict with load scalability as compression takes time.

Data structures influences both load and space scalability. Data structures such as arrays or similar structures that utilizes address space have a finite amount indices that are available for use. This leads to there being an inherent constraint on the possibilities of growth. It is possible to circumvent this problem by operating with dynamic arrays as opposed to static ones. Every time a new element is added, the current array is being replaced by a new one that a size equal to the size of the current array plus one. Every time this new array is being created it has to be populated with the elements from the old array, which is expensive. As the amount of elements in the array increases, the time taken to populate the new array will reach intolerable levels.

A system is considered space-time scalable if the underlying data structures support algorithms that perform well regardless of system size. Again, the choice of data structures heavily influences to which extent a system is regarded space-time scalable. For a system to be space-time scalable, its access algorithms should not exceed a complexity measure of $log_2(N)$. A system that adopts a linked list structure will exhibit linear performance, and thus would not meet the requirements for scalability. A data structure such as a binary search tree or a hash-table would meet these requirements as the complexity measures are at the threshold or better.

Structural scalability can be seen as the properties of a given system or architecture that foster further development. A system with an architecture that does not impede the growth of a number of objects that it encompasses through its established implementation interfaces is seen as a structural scalable system.

One should strive to ensure that there is no discrete bound value where scaling beyond this point is impossible. An example of this scaling problem is that of the IP-

addressing scheme of the Internet. In 1970 it was decided that addresses consisting 32 bits were sufficient to supply the worlds demand for IP-addresses. This scheme was known as the IPv4 addressing scheme. We now know that this was a defeatist prediction. As of July 2005, 19% of the available IP addresses has been consumed. One may perceive this problem as a both one of load-scaling and that of structural-scaling. An ongoing project to fix this inevitable bottleneck is the IPv6 project, where the amount of bits used in addresses are extended to 128 bits. [7]

If parallelism is to be regarded as adequate, it has to cater for the usage of multiple processors to perform tasks asynchronously. That is, tasks that do not need blocking should be carried out immediately by an idle processor. Single-threaded systems fail to do this and can be said to suffer from poor load scalability that stems from poor structural scalability.

Approaches to scaling

There are two approaches to scaling described by Coulouris et al: *horizontal* and *vertical scaling*. Horizontal scaling is the act of adding more nodes physical to a system in order to improve performance. Vertical scaling is to manipulate the resources of already existing physical nodes in the system.

Improving scalability

Ways to deal with performance bottlenecks related to scaling are: data replication, data caching and the deployment of multiple servers.

Transparency

Transparency is the process of masking the underlying complexity of a system, effectively abstracting potential multiple heterogeneous components such that the system can be perceived as a single coherent unit as opposed to many separate components intertwined with varying requirements and interfaces. We will cover *eight* different types of transparency; access, location, concurrency, replication, failure, mobility, performance and scaling transparency.

Access transparency

The system makes no distinction between local and remote resource access. One utilizes the exact same operations for both modes through a hidden interface that may act as a proxy between remote resources in the network.

Location transparency

Remote resources may be located on different physical locations, potentially being far away from the calling object. Any operation that is performed on a remote location will not be affected of the fact that the resource is on another physical node far away. All operations should be performed seamlessly.

Concurrency transparency

The extent to which multiple processes working on the same resources can continue so without any interruption.

Replication transparency

The use of replicas in systems to provide better performance and fault tolerance. To the end user there should be no difference in accessing a replicated resource from that of an original resource.

Failure transparency

To allow the system to continue functioning despite software or hardware components failing.

Mobility transparency

The abstraction of moved code, for example by traveling to a new location.

Performance transparency

Performance transparency refers to the ease of reconfiguration of a given system; to which extent one may modify it by adding new modules or changing existing ones. This is also related to the concept of *Openness*, which is previously described.

Scaling transparency

Enables the act of scaling up the system efficiently without changing the underlying architecture.

3.2 Memory hierarchy

The contents of this section and the following on caching is derived from chapter 2 (memory hierarchy design) and appendix b (review of memory hierarchy) of the textbook "Computer Architecture: A Quantitative Approach" [12].

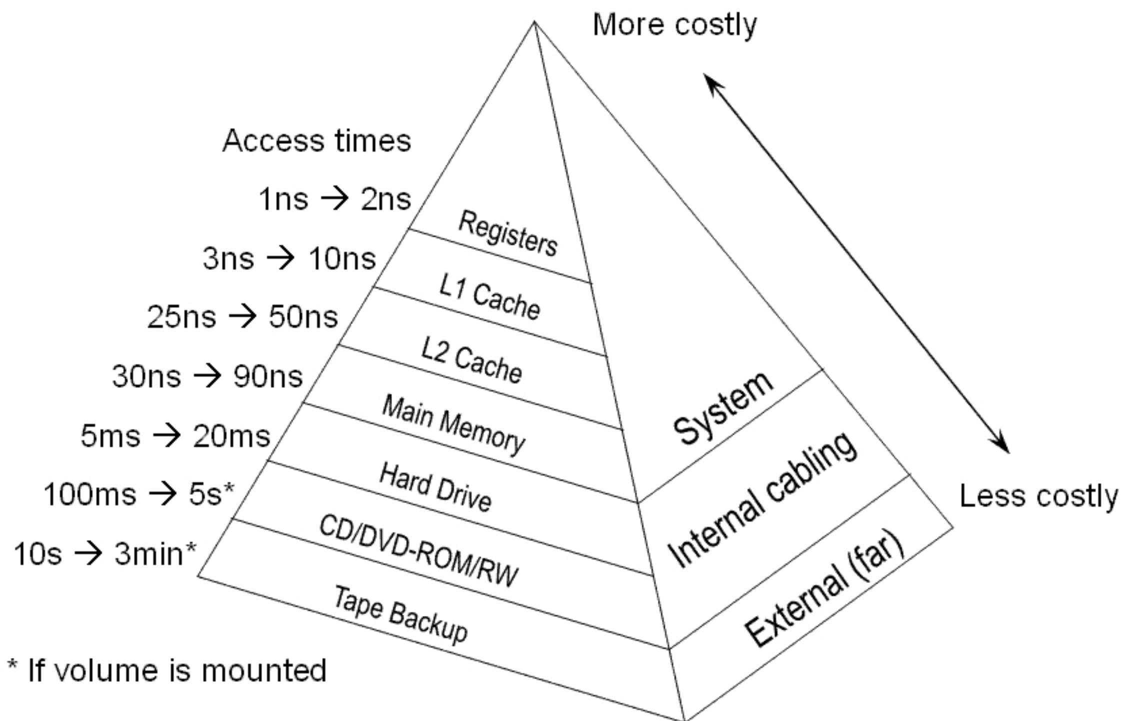


Figure 3.1: Illustration of a traditional memory hierarchy [8]]

In an ideal world, memory would be fast and cheap. However, this is not the case. There are different types of memory, each with different attributes. When choosing a memory type, there will always be a compromise to be made. One can view memory technology at a spectrum; *The faster the memory, the smaller, more expensive and more power hungry it becomes.* What one refers to as "fast memory" is called **static memory**, abbreviated **SRAM**. It is this type of memory that the **CPU** is made of. Even with state of the art **cooling technology**, one cannot cope with the power requirements that come with stacking big quantities of SRAM on a single chip. This leads to the inevitable fact that it simply is not feasible to have big quantities of SRAM. Instead, one combines different types of memory to form a **hierarchy** of memory.

At the top of the hierarchy we have the CPU, in the middle we have the **main memory** which is made from a slower but cheaper memory type called **dynamic memory (DRAM)**, and at the bottom of the hierarchy we have the secondary storage, which typically consists of **hard disk drives** (HDD). Static and dynamic RAM are both **volatile**, whereas he secondary storage is **nonvolatile**. A volatile medium will flush all data that is stored during runtime when it is powered off. In contrast, a nonvolatile medium may store data for further use in the future. In between the CPU and main memory there exists **cache memory**, which is further divided into levels.

The cache memory works as a filter to reduce the traffic on the **memory bus** that is located between the CPU and main memory. It is an essential part of the memory hierarchy. The goal of a memory hierarchy is to provide data to the registers of the CPU as quickly as possible. This is achieved by *caching frequently used data*. Whenever an instruction is needed by the CPU, it will first check its caches. If it fails to find the data at the first cache in the hierarchy, it will continue searching further down in the memory hierarchy until it reaches the secondary storage. At this point the main memory will have to issue a **page fault** to fetch the necessary instructions from secondary storage, and then proceed to cache the instruction in the CPU caches. If, however, any data is found in the volatile parts of the hierarchy, it will fetch that data directly.

3.2.1 Caching

A **cache** is a component that temporary stores frequently accessed data derived from another source so that it can be retrieved faster in the future. It is assumed that stored data will be frequently accessed in the future. The most known cache is the CPU, which has it's own dedicated on-chip caches. It increases system performance in two major ways: it reduces memory latency by minimizing the amount of CPU stalls due to memory requests, and it attempts to keep memory bus pressure low by the reduction of data throughput. A key issue with the modern multicore processors is that the memory bandwidth is shared between the cores, leading to even less **memory bandwidth**, and even more need for ways to reduce bus pressure. In a distributed system it may enhance performance as the amount of packages being transmitted will be reduced. Instead of making invocations towards a remote server every time a file is requested by the user, one may first perform a search on the local cache to see if it is present. Any reduction of **I/O**, whether it be disk accesses or network load, will lead to performance gains.

Cache motivation

A key influence on the development of caches is the fact that certain memory addresses will be accessed frequently, in computer science this concept is called **locality of reference**. A cache should take advantage of this concept, as well as having some sort of **branch prediction** mechanism in place.

Spatial Locality

When accessing a specific memory address, there is a high probability of having subsequent accesses to the memory block that the address resides in, or neighboring memory blocks. This principle is the motivation behind the implementation of the **cache controller**, and the **cache prefetcher**.

Temporal Locality

A given sequence of references to memory locations will be referenced multiple times during the lifetime of a program, and references to its successors will also be made. Multiple accesses of the same set of elements in a data structure may contribute to better temporal locality, and thus better cache performance.

Branch prediction

Branch prediction is the process of identifying the probability that a given cache line, or neighbouring cache line will be accessed frequently. Modern CPU design incorporates a on-chip hardware component referred to as the **hardware prefetcher**. The duty of the prefetcher is to figure out which data instructions that is to be called next. It does so by calculating in real-time the probability that a given set of instructions are going to be accessed in the near future based on access patterns. The prefetcher learns from its accesses, and *becomes more efficient at calculating as the frequency of data being accessed from the same set of addresses increases*. Therefore it is beneficial for the data to be arranged in sequence, to exhibit locality as described above. Ways to do this is to avoid using dynamic memory allocations, and instead rely on stack structures such as arrays. The developer has to make a compromise between efficient data structures and efficient data access patterns.

Multi-level cache

As previously mentioned, the caches on a computer are organized in layers. The top layers are small, fast and expensive, whereas the bottom layers are bigger, slower and cheaper. The fastest level cache is commonly referred to as **L1**, and there typically exists a secondary cache named **L2** and optionally a third cache that has the acronym **L3** or **LLC** (last level cache). The closer that a cache is located in relation to the registers, the faster it will perform. It is also possible to view the main memory as a type of cache that is significantly faster and more expensive than that of data stored on the disk. The main memory is commonly referred to as dynamic memory (DRAM), whereas the CPU is comprised of static memory (SRAM). Static memory is faster than dynamic memory, but is also more expensive. Caches are bridged by memory buses. Figure 3.2 provide an illustration of a two-level cache.

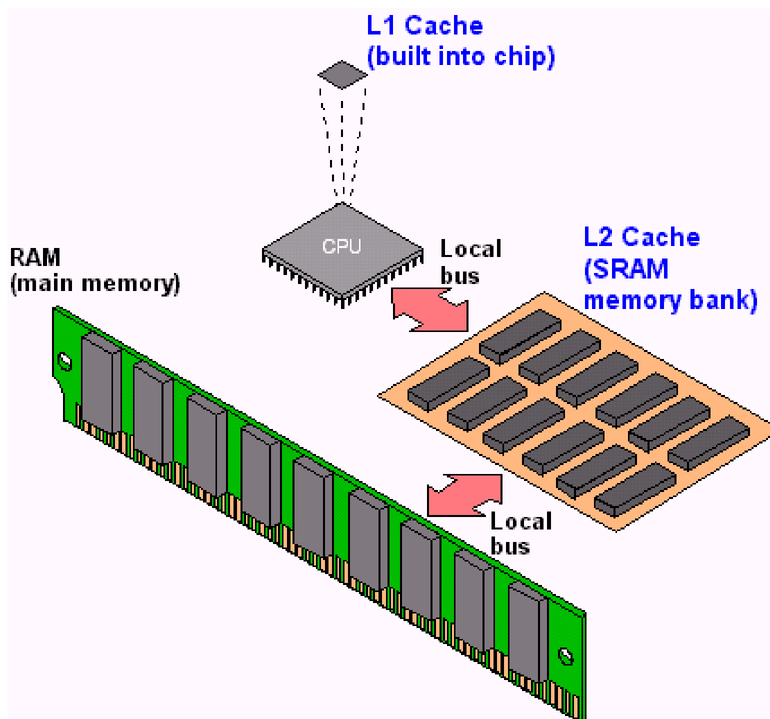


Figure 3.2: Illustration of a two-level cache [10]

Key cache concepts

Cache hit

Whenever a certain data element is to be accessed by the CPU, it first looks in the L1 cache for the specified instruction. If it is found a cache hit occurs.

Cache miss

When the specified data element is not found in the L1 cache, it will check the layers further down the memory hierarchy, specifically the L2 and the LLC (last level cache). If the requested data is not found in the LLC, a cache miss has occurred. Whenever a cache miss occurs, the CPU has to fetch the instruction from the main memory, and load it into the L1 cache for further use. This is initially done by the cache controller, and succeeded by the prefetcher (as described on the previous page). If the specified data item is not located in the RAM, a page fault occurs, and the item has to be fetched from disk.

There exists four types of misses; namely that of *compulsory, capacity, conflict* and *coherence misses.*

Compulsory-miss

The initial cache search will always fail; this is because nothing has been stored in the cache. This is also referred to as a *cold-start miss.* It follows the same logic as the page fault concept in the RAM.

Capacity-miss

A capacity miss occurs when the cache is full. This is bound to happen in any system, and it is therefore vital that one has sufficient algorithms for data replace-

ment in place. An example of such an algorithm is the LRU (least recently used), where the data items that are being accessed the least is being replaced by that of the new data item.

Conflict-miss

Conflict-misses are due to clash in indexes between the memory block and the cache line, the address that was requested is already in used. This type of miss is commonly seen in cache layout strategies based on direct mapping.

Coherence-miss

Coherence-based misses are due to the invalidation or corruption of accessed data, which is being carried out by another process.

Cache performance

Cache performance is influenced by three factors: *miss rate*, *miss penalty* and *hit speed*.

Miss rate

Cache miss rate is to which extent the cache fails to identify an elements specified by the CPU. The cache hit rate can be calculated as follows: (1 – cache miss rate).

Miss penalty

Cache miss penalty is referring to how much time it takes to copy data from memory or disk into the cache.

Hit speed

Cache hit is referring to the speed of cache look-ups.

Average memory access time (AMAT)

The average time it takes to access a certain data element X can be described with the following equation: $AMAT = Hit\ Speed + Miss\ Rate + Miss\ Penalty$

Cache layout strategies

There are three main strategies used for designing cache algorithms, commonly referred to as layout strategies. The most common layout strategies are *direct-mapped*, *set-associative* and *fully-associative*.

Direct-Mapped

In a directly mapped cache, the memory blocks can be mapped to only one cache line. The advantage of this strategy is that searches are magnitudes faster than that of an associative-based strategy. The disadvantages, however, are that mapping conflicts may lead to increased miss-rate. In other words, the conflict-misses will increase. This layout may also be referred to as a one-way associative cache. The number of bits used for the index in a direct-mapped cache is $log_2(N)$, where N is the amount of cache lines. In a direct-mapped cache with 10 cache memory locations (0 through 9) and 40 main memory locations (0 through 39) available, how does one make sure that all 40 memory addresses are mapped to cache location? A simple solution to this problem is to use an indexing algorithm that utilizes the modulo (%) operator. To identify the cache index of a given memory location N, one may use the following formula: $memloc =$ (N % 10). For the memory location 20, this equates to an index of 0, since there is no remainder after performing the calculation $20/10$.

But one cannot rely solely on the modulo operator. The reasoning for this is that it will eventually lead to collisions, especially if the algorithm is crude. To exemplify this; both 37, 17 and 7 equates to an index of 7. A way to counter this used in caching is the so-called tag. The tag is an added identifier, which is calculated with a function in a similar manner that of the index. The index and the tag are used in conjunction to determine the cache address of a given memory address.

Fully-Associative

In contrast to a direct-mapped scheme, an associative cache can map any memory address to any cache line. When inserting data, the memory address is compared to all cache line tags in parallel. Whenever a cache-miss occurs, one must either place the data in a valid index or to swap with an already existing data element. The data element to be swapped depends on the eviction policy, which are guidelines for which data elements one chooses to swap out in the light of a full cache.

Set-associative

Set-associative is a compromise between the complex fully-associative and that simplistic direct-mapped cache layout strategy.

Summary

Computer memory is organised in layers, where the top layers are faster, smaller, more expensive, and generate more heat. Because fast memory is expensive, one can only put so much on a given computer. Caching is the act of temporarily storing frequently accessed data. Cache memory is placed on the limited, fast parts of memory (SRAM).

3.3 Graph Theory

Graphs are mathematical concepts that are made up of a collection of points, called vertices that are connected by lines, called edges [26]. There is a wide spectrum of fields that employ graph structures to investigate variables of interest. The most common usage for graphs are in computer networks, planning and scheduling of projects and interactions between actors in social sciences. It should be notes that we will exclusively cover what is deemed relevant to this thesis, which implies that redundant information is omitted.

Graph Constituents

A graph is made up of two sets called vertices and Edges. Each element contained in the Edge set is a pair consisting of two elements from the vertices set. These pairs acts a relation between vertices. For instance the edge [V1,V2] implies that the point of origin is that of the vertex V1, and the destination is the vertex V2. Vertex V1 may reach V2 through it's corresponding edge that describes this relationship.

Graph types

Graphs may be directed, or undirected. In directed graphs vertices do only have outbound edges, that is, edges that contains merely the destination vertex. It is thus not possible for any destination vertex, such as V2 in edge set V1,V2 to reach V1. V1 is however able to reach V2.

On the other hand, in undirected graphs - vertices are free to traverse back to their origin vertex. This means that in the edge set [V1, V2], V2 may now reach its origin vertex V1.

A directed acyclic graph (DAG) is a directed graph that contains no cycles. A tree structure is a special kind of DAG.

An example graph

To illustrate this concept, let the sets V and E comprise our graph G = V, E. The V set contains the vertices 1, 2, 3, 4, 5. The E set contains the relations between vertices [1,2],[1,3], [3,5], [5,4]

As can be observed, the V set contains all the possible vertices in the graph. The connections between these are described in the E set. Below is a graphical illustration of this example graph.

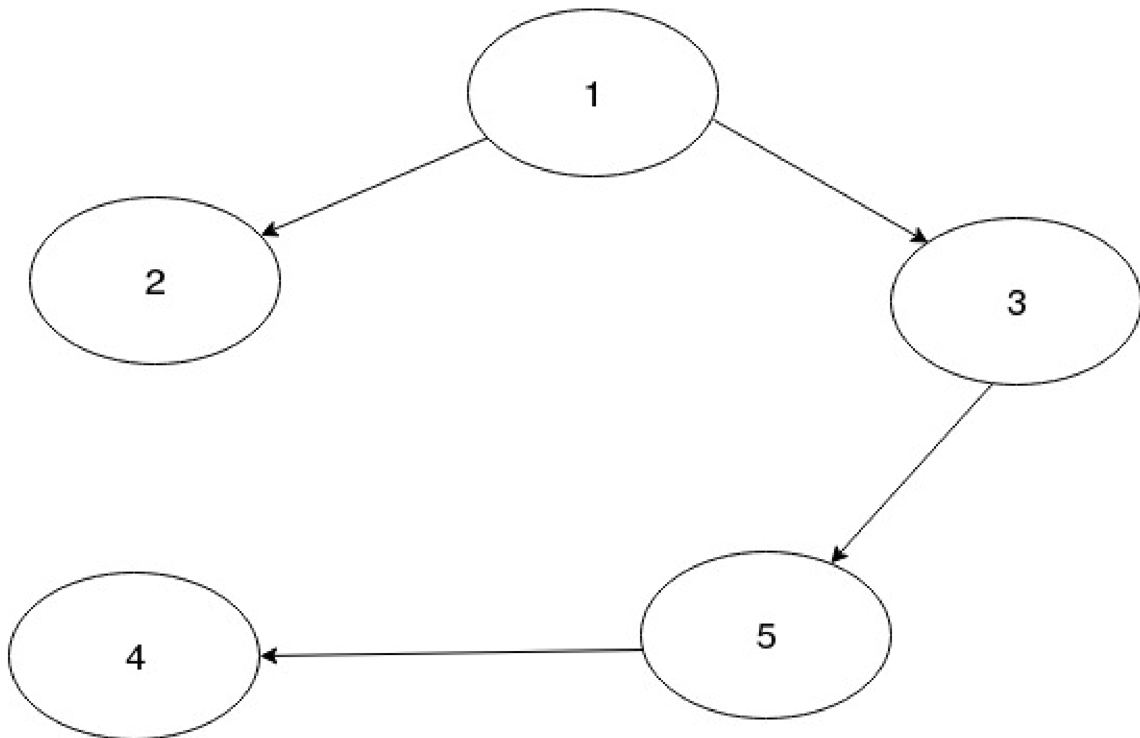


Figure 3.3: Illustration of our example graph [24]

Depth-First Search

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the vertices by traversing into the descendants of a given vertex, until it reaches a leaf vertex. When a leaf vertex has been encountered, one is to backtrack. Backtracking implies returning to the previous stack calls in the recursion. The algorithm terminates either if all vertices have been visited or if the vertex of interest was found.

```
recursiveDFS(currentVertex, targetID)

    while currentVertex.getID ≠ targetID
          do currentVertex ← currentVertex.visit
             for each children ∈ currentVertex.getEdges
                 if children ≠ visited then
                    do recursiveDFS(children, targetID)
```

Figure 3.4: Recursive depth-first search [24]

Complexity of DFS

The complexity associated with DFS is $O(|V| + |E|)$.

Breadth-First Search

BFS is a traversing algorithm where one starts traversing from a selected vertex, and traverse the graph layer-wise - thus exploring the neighbouring vertices that are connected to the given vertex. One is then to move towards the next-level of neighbouring vertices. One typically required to traverse the graph breadth-wise as follows: first move horizontally and visit all the vertices of the current layer, then move on to the next layer and repeat the process.

```
BFS(rootVertex)

    Q <- Queue.create
    Q.enqueue(rootVertex)
    root <- root.visit

    while Q ≠ empty
          do currentVertex ← Q.dequeue
             for each children ∈ currentVertex.getEdges
                 if children ≠ visited then
                    do Q.enqueue(children)
                       children ← children.visit
```

Figure 3.5: Breadth-First Search [24]

Complexity of BFS

The complexity associated with BFS is $O(|V| + |E|)$.

3.4 Cloud computing

Cloud computing is a term used to describe services that provide access to configurable, on-demand computing over the Internet [9]. Computing resources are commonly organized in a *pools*, where clients may be provisioned a certain amount of the given resources at any given time. Resources that may be provided by the cloud service may be computing power or storage capacity, bandwidth and latency. Cloud computing may be compared to an electricity grid, where resources may be increased or decreased depending on needs. The advantages of cloud computing is that one may alleviate the upfront costs that are associated with deploying a service. This allows companies to focus on their applications as opposed to worry about the network infrastructure, and thus get their business up and running faster. Maintenance is commonly provided by the 3rd party data center, and scaling capabilities are excellent [20]. The pricing model used in cloud computing is referred to as "pay as you go"; each month one is billed a sum that corresponds to the use of resources. As such, it is vital to carefully monitor and restrict resource consumption for clients.

3.4.1 Cloud hierarchy

The services may provide resources that are located far from the client, or resources that reside on nearby servers. One may view the resources in the cloud as organized in a *hierarchy*, similar to that of the memory hierarchy. The further away the resources are, the better the performance. A service that provides a storage solution may be located in a third-party data center far away, and will exhibit excellent computing power, storage capacity and bandwidth. However, the latency from this very service will also exhibit high latency towards that of the User and L1 Cloud. If the client was to perform frequent invocations towards that of the L3 cloud, it would as expensive as the latency between them is significant.

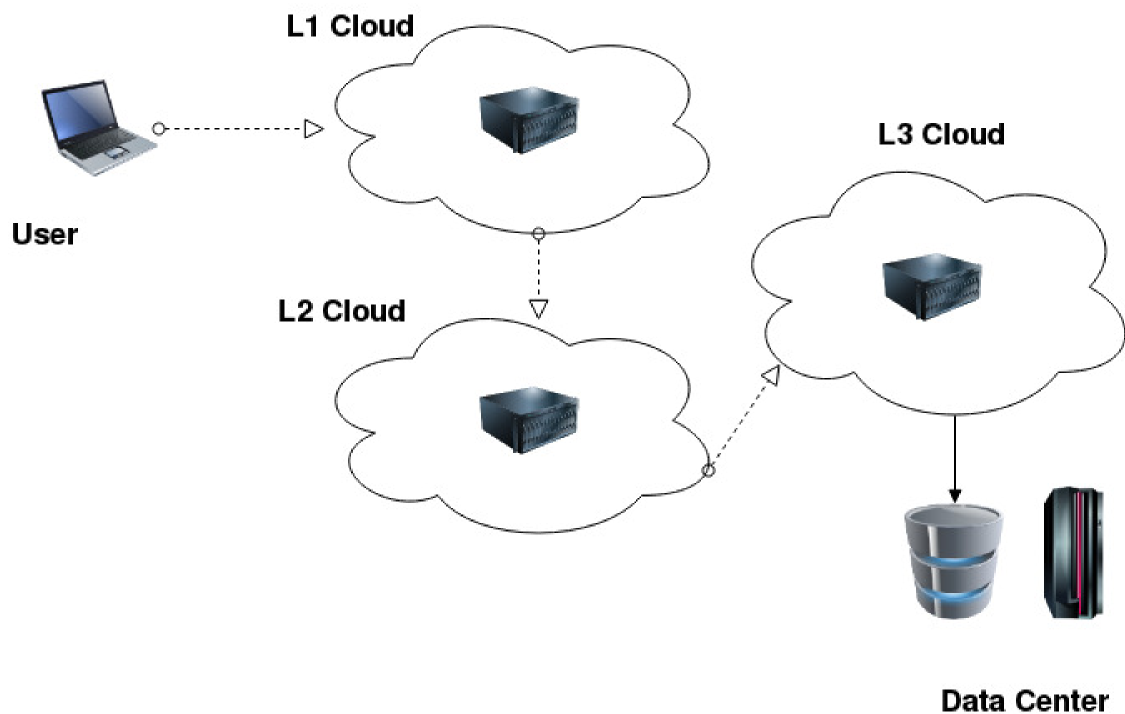


Figure 3.6: Illustration of the cloud hierarchy [24]

3.4.2 Near-far cloud

As previously touched upon, data centers are located far from the client, and are equipped with powerful resources. However, this implies that latency towards the client will suffer. If the client is to frequently request data from the data center, the network-transmission overhead will quickly become a bottleneck in the system. A solution to this is have a *near cloud* that is between the *far cloud* (data center) and the client. With this structure, the near-cloud may retrieve data from the far-cloud, process the data, and then transmit the results to the client. This comes in handy, because the client has limited computing power and it's latency towards the data center is too big. The client's latency towards the near-cloud is minuscule, thus minimizing overhead costs associated with network transmission. One may view the client, near-cloud and far-cloud as being in a multi-level hierarchy, similar to that of multi-level caching. The L1 cache would be the client, L2 the near-cloud and LLC the far-cloud.

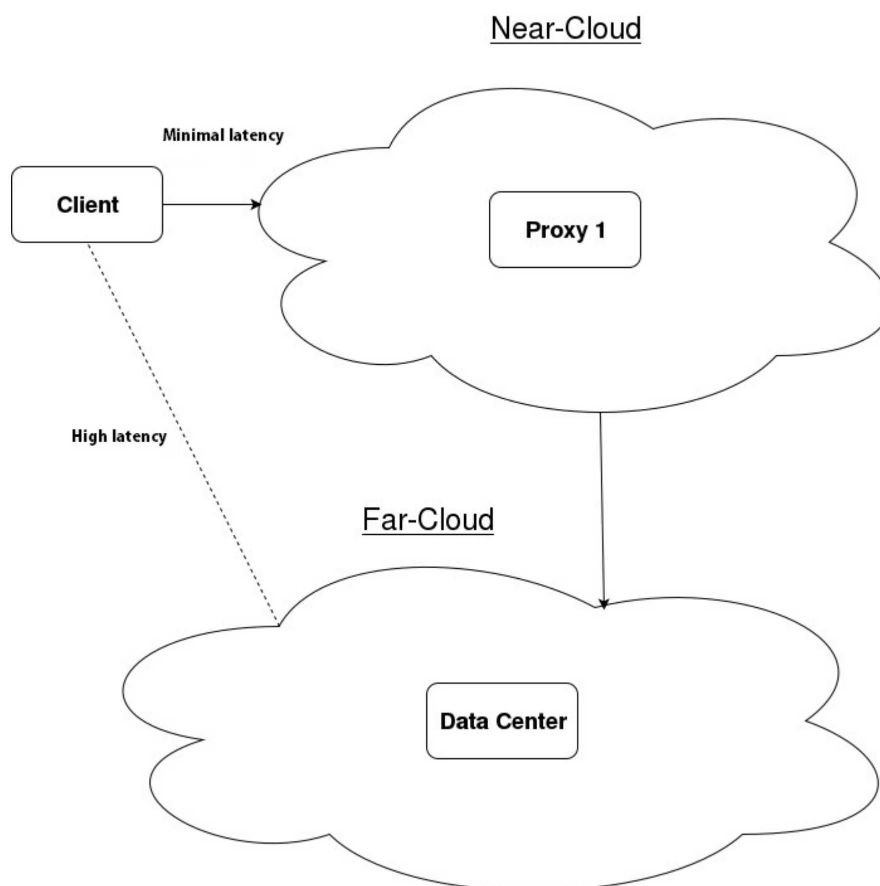


Figure 3.7: Illustration of the near-far cloud [24]

3.5 Emerald

Emerald is a distributed objects programming language that was developed in the 80's by Eric B. Jul, Andrew P. Black, Henry M. Levy and Norman C. Hutchinson. The goal of the language was to simplify the design of distributed applications, and to make distributed programs more efficient. This was done by developing a language that was centered around the concept of *object mobility.* [3], [13].

3.5.1 Object

Objects are the key constituents of the language. Since object mobility is a key aspect of emerald, objects tend to be of higher complexity than that compared to other languages. For instance, it is fully possible, and sometimes preferred, to have entire services contained in one object. This is because in Emerald, you may move any object to Nodes in the system. A Node is the logical representation of a physical server somewhere in the system. This makes migration of services less tedious.

3.5.2 Object mobility

Object mobility is the capability of a given object to travel through a system, taking advantage of distribution of data and state-changes. Emerald supports *fine grained* object mobility [15]. Fine grained refers to the size of objects, where in Emerald the units of mobility may be data objects and processes. Object mobility encompass both the act of process migration and that of data transfer.

3.5.3 Data Migration

Process migration is a crucial part of Emerald. It is the act of moving a live process to another node, and performing the required amount of translations between the node of the old process and the new one. A node is an identifier used to provide an abstraction of physical machines in form of an object. In Emerald the process-migration translation is done seamlessly, the running process is *ignorant of*

its location and unaffected by the move]. This gives Emerald strong location and mobility transparency, as described in section 3.1.2 on Transparency

Emerald is equipped with five location-dependant operations, termed mobility primitives to aid in object-mobility: locate, move, fix, unfix and refix [14].

Locate

Locate attempts to pinpoint the physical location where an object currently is resident. The return value of this operation is assigned to a Node object that is to represent this very physical location. One has to be vary of the fact that in a distributed system, the location of objects are subject to frequent change. The application programmer must not under any circumstance become naive in the sense that it is expected that the location of object X remains the same 1 second into the future.

$$currentNode \quad \leftarrow \quad locate \quad x \tag{3.1}$$

Move

The move primitive attempts to move an object to the designated location. The expression [move x to location] indicates a desire to relocate the object associated with parameter x to the location associated with the parameter Node. One should note that move invocations does not guarantee successful migration of objects. It is merely a hint to the kernel that the object is to be relocated to the specified node. If the object currently is under effect of a fix invocation, nothing will happen.

$$move \quad x \quad to \quad location \tag{3.2}$$

Fix

The big brother of move, with stronger semantics. The fix statement indicates that the specified object is guaranteed to relocate to the given node, and that it is to stay

there until unfix or refix has been called on the object. If the object has already been fixed at another location, one will be informed of this by an error message. The advantage of using fix is that it provides a more reliable means of object migration, however this comes at the expense of performance. If reliability is not of absolute concern, or if objects will exhibit frequent relocation one is to rather utilize the move statement than that of fix.

$$fix \quad x \quad at \quad location \tag{3.3}$$

Unfix

The unfix statement will unlock an object that has previously been locked in place by the fix statement. It is now free to migrate to another location.

$$unfix \quad x \tag{3.4}$$

Refix

The refix statement will act as both as an invocation of unfix and an invocation of fix on the target object. The result is that the object X is now relocated at the new node associated with the variable location. It cannot relocate unless unfix or refix is being called again. The syntax is as following:

$$refix \quad x \quad at \quad location \tag{3.5}$$

3.5.4 Language Constituents

```
const Kilroy ← object Kilroy

        process

          const origin ← locate self
          const up ← origin.getActiveNodes

          for e in up

                const there ← e.getTheNode
                move self to there

          end for

          move self to origin

        end process

end Kilroy
```

Figure 3.8: "Kilroy", a classic emerald program

Figure 7.8 provides a sample Emerald program. It consists of a **process** clause, **constant** variables, a **for-loop** and some operation calls. As previously mention in section 3.5.3 on data migration, the expression "locate self" will return the node in which the current object resides. The function "getActiveNodes" returns a NodeList, which is a list of all active nodes in the system. Our object Kilroy is to travel to every node in the system, and upon completion travel back to its orgin node. The process clause indicates that a lightweight thread has been started, these are necessary in order to perform our data migration operations.

For more information on the constituents Emerald programming language, one should consult the language report for a full reference [14].

Attached

The attached keyword indicates that the given associated object is also to be moved if the the object it resides is moved. That is, if object X has a reference to object Y, and that reference has the attached keyword in front of it; object Y will also be moved to the designated location if object X is to be moved. This is useful for recursive data types. Transmission of recursive data structures such as linked lists, trees and graphs are greatly simplified. To transfer an entire list, tree, or graph all one has to do is to move a single object. This works because a list is nothing more than a recursive chain of pointers to objects of the same type. If these links are to be marked as attached, all connected objects are to be transmitted. Figure 3.9 and 3.10 provides an example of a List programmed in Emerald that utilizes attached.

```
const Pointer <- Class Pointer[value:Integer]

        attached var next:Pointer
        attached var element:Integer <- value

        export operation pointer_setNext[nextPointer:Pointer]
                next <- nextPointer
        end pointer_setNext

        export function pointer_getNext -> [retPointer:Pointer]
                retPointer <- next
        end pointer_getNext

end Pointer
```

Figure 3.9: Class Pointer

```
const List <- Class List

        attached var head:Pointer

        export operation list_transmit[location:Node] -> [retPointer:Pointer]
                move head to location
                retPointer <- head
        end list_transmit

        export operation list_add[value:Integer]

                if(head == NIL) then
                    head <- Pointer.create[value]
                else

                    var temp:Pointer <- head

                    loop
                            exit when temp == NIL
                            temp <- temp.pointer_getNext
                    end loop

                    temp.pointer_setNext[Pointer.create[value]]

                end if

        end list_add

end List
```

Figure 3.10: Class List

The head of the list is recursively connected to other points through their variable "next". Note that that all variables have been marked as **attached**. If the attribute "next" in class pointer was not to be marked as attached, these objects would not be included in an event of migration. Any calls to these objects would turn into *remote invocations*, as they do not reside on the node.

3.5.5 Emerald Type System

A key goal when developing the Emerald type system was to ensure that it supported compile-time type checking as well as being based on classification rather than on implementation [3]. The type system was initially inspired by the concept known as operation-centric protocols, which was an idea derived from Smalltalk [11].

Purpose of type system

Instead of merely guarding object data against unintended use, a type system may serve other noble causes as well. For instance, it can provide classification, which may ease the development of complex systems by providing means of abstraction through types. It also supports early error detection and aid in performance optimization's by the compiler.

The idea that all systems should be extensible, called the **open world assumption** was a popular idea back the day when the type system was under development. Of course, nowadays it is taken for granted that systems have to be extensible, as covered in the distributed systems section on the concept of openness. A consequence of the open world assumption is that the program has to cope with the fact that there will exist objects that didn't exists at the time the program was written. This implies that there could potentially exists objects whose types were unknown at the time when the program was initially written.

The term **concrete type** was coined as a way to refer to the set of operations that was understood by an object. The term **abstract type** was used to describe the set of operations being specified in a programming language syntax. The duty of the type system is to check whether or not the concrete type in question supports enough operations in order to be used in the context of the abstract type in question. The term **conformity** is being used to ascertain to which degree an object X of type U may act in the context of type V. Conformity ensures that a concrete type understands a superset of the required operations being specified at the abstract type . This implies that the operation names, parameter types and return value types has to be identical.

The decision to implement a conformity-based type system rather than one of the contemporary type systems based on sub-classing, such as Simula, had to do with the fact that Emerald was a language *tailored for distributed objects*. In a distributed system, the important questions are not about the implementation of an object, but about the set of operations that it implements. As previously mentioned in our section on distributed systems, they are volatile, objects may relocate and are being susceptible to change upon runtime. Furthermore, a conformity-based type system may aid in developing **loosely coupled** systems.

Type equality, which existed in sub-classing based type systems, was not just redundant, as a matter of fact they were deemed counterproductive [3]. If object X wanted to act in the context of type T, it had to conform to its required operations. However, it could not be equipped with any additional operations besides those that type T describes. Alas, it was forbidden for its set of operations to be a superset that of type T.

The idea of conformity revolves around the notion that an object X of type T, conforms to that of the type U. This implies that the type T has to contain a superset of the operations contained in type U. The symbol

$$o> \tag{3.6}$$

is used to to illustrate that the type on the left contains a superset of the operations which are contained in the type on the right, written

$$T \quad o> \quad U \tag{3.7}$$

Type checking and error handling

The open world assumption leads to the inevitable fact that under come circumstances, type checking had to be performed at runtime. The reason being that there may exist objects at runtime that were not accounted for at compile-time. Compile-time type checking implies that one guarantees that there is no type errors present at the time of compilation, however there are exceptions to this rule. What if the object will not be available until runtime? In a volatile distributed systems where objects constantly are being produced, removed, changed and relocated it may deem counterproductive to apply compile-time determinism. In theory, one cannot guarantee that there will exist objects at runtime that were not accounted for at compile-time.

The solution to this is to combine compile-time checks with run-time checks. If there is sufficient information available in order to ascertain that a given object is certified to act in the requested context, the object is deduced to be type-correct and the request for behaving according to the specified context is granted. However, if the information present at the time is deemed inadequate, the type-checking is deferred to run-time. The expression "view as" was invented as a way for the programmer to explicitly defer type checking to run-time:

$$newType \quad \leftarrow \quad view \quad Type1 \quad as \quad Type2 \tag{3.8}$$

will run a conformity check on type 1 to make sure that it conforms to that of type 2, and if successful, it will be able to invoke operations from Type 2 through the returned reference newType. The "view as" expression is inspired by the "inspect" statement derived from Simula 67.

Transparency in Emerald

Emerald was created with transparency in mind. It comes with from the get-go support for location and mobility transparency, and may with ease be configured to support access, replication and failure transparency. Similarly to CORBA and Java RMI [7], location transparency is supported by hiding abstraction associated with remote object. There is by no perceived difference between calls made to local objects and calls made to remote ones. This results creates an if the whole system resides on a single computer. In other conventional languages such as C, one has distinct classes that are tailored for the respective nodes. This means that each node in the system has some code that is unique and bound to that very physical location. In Emerald, however, one may use the exact same piece of code on all computers.

This level of abstraction, although confusing at first, greatly simplifies the development of distributed applications. Mobility transparency is closely related to location transparency, and is key to Emerald. The complexity associated with the migration of objects is hidden. Objects may be moved around in the system effortlessly, without the application programmer being notified.

Mobility transparency is supported by the system remaining unaffected by the migration of objects. If an object or service were subject to cross-node migration, one would not now of it. The combination of access, location and mobility transparency gives us a system with a high degree of abstraction.

This high level of transparency has both advantages and disadvantages. The advantages are that it is fairly intuitive and user-friendly, since complex "under-the-hood" sorcery are taken out of the equation. This way, it simplifies the process of developing distributed applications, minimizing the amount of code required. However, all this transparency have made the inner workings of the system harder to fathom. Debugging may prove difficult. This is especially true in systems spanning multiple nodes. It is mandatory that the code-base is structured and that one is utilizing location-dependant operations liberally.

3.6 Planetlab

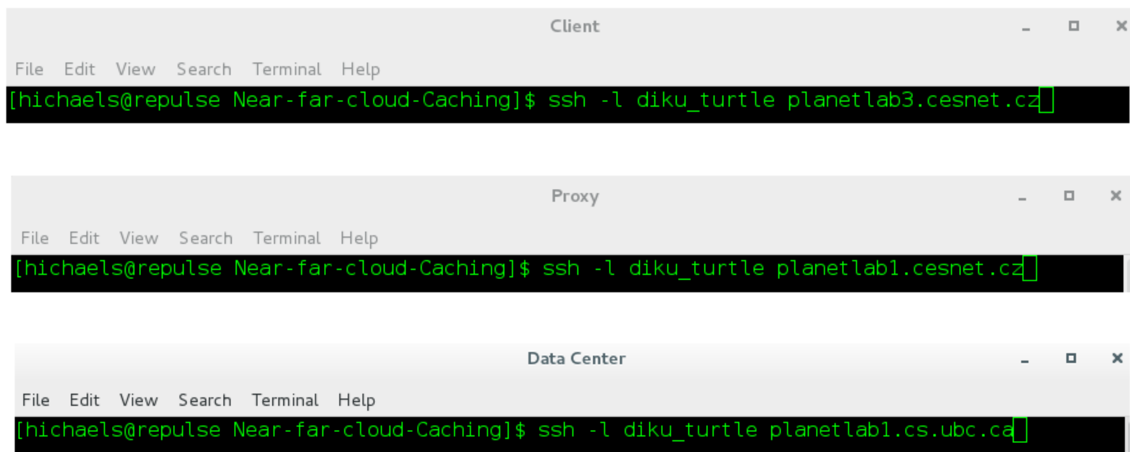
PlanetLab is a global research network that supports the development of new network services. Academic institutions and industrial research labs use it to develop new technologies for distributed systems [2].

Planetlab consists of a cluster of nodes, distributed across the world. One utilizes *secure shell* to get access to these nodes. In this thesis Planetlab will be used to distribute data and functionality among nodes located at various locations in the cluster.

Below is an example of how to connect to the Planetlab network. We are utilizing the same three nodes used in our evaluation.

3.6.1 Step one

We are connecting to three different Planetlab nodes by the means of secure shell. In this example, the associated Planetlab slice is called "diku turtle".



3.6.2 Step two

Once connected, we perfom a hostname check to verify that we are indeed on the correct nodes.

Client

File Edit View Search Terminal Help

```
[diku_turtle@planetlab3 ~]$ hostname
planetlab3.cesnet.cz
```

Proxy

File Edit View Search Terminal Help

```
[diku_turtle@planetlab1 ~]$ hostname
planetlab1.cesnet.cz
```

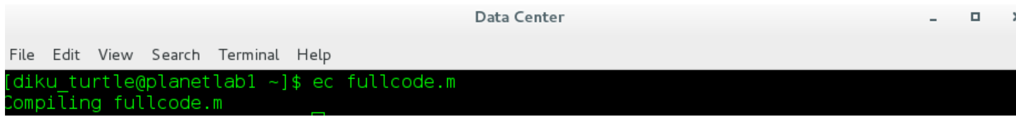
Data Center

File Edit View Search Terminal Help

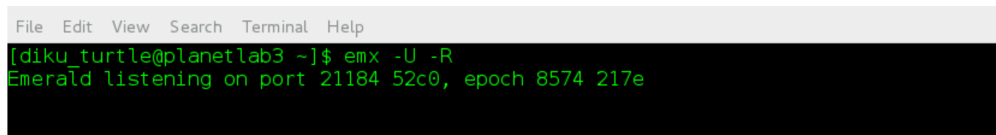
```
[diku_turtle@planetlab1 ~]$ hostname
planetlab1.cs.ubc.ca
```

3.6.3 Step three

We now give an example of running a distributed application.



First, one has to compile our emerald program "fullcode.m" on the data center node.



We then move on to the initial step of our scheme. The client has to perform the call "EMX -U -R" which indicates that it is now an active node on the Planetlab system.

The proxy does the same, but now one also adds the clients address and port number.

Similary to the proxy, one adds the hostname and port and runs the compiled program by calling "emx fullcode.x" ".x" is the executable file that has been generated from our ".m" file.

```
File  Edit  View  Search  Terminal  Help
[diku_turtle@planetlab1 ~]$ emx -U -Rplanetlab3.cesnet.cz:21184 fullcode.x
Emerald listening on port 16622 40ee, epoch 27765 6c75
```

3.7 Summary

In this chapter we have presented the theoretical underpinnings that motivated our thesis. Distributed systems, memory technology, graph theory, the Emerald programming language and Planetlab have been covered. We have also shown how these topics intersect.

CHAPTER 4 PROBLEM STATEMENT

4.1 The issue

In our motivation chapter we identified the main factor that contributes to a restriction on performance in a distributed system; the slow speed of light. The speed of light influences latency. In distributed systems where nodes are located far from one another (exhibiting high latency) and packet exchange between these nodes are *frequent*, performance degradation will occur. What we wish to achieve is twofold; to somehow minimize latency from the client to that of the entity that carries out computation, and to limit the amount of packages that is to be transmitted between these entities.

4.2 The solution

Latency can be reduced by incorporating the near-far concept, as previously described in our background chapter. In a classic client-server architecture, the client is making invocations to that of the server. Each call is potentially expensive due to the very fact that there exists latency between between the two that correlates to their difference in geographic location. The near-far concept may be utilized to implement a system where there exists an *intermediary entity* between that of the

client and the server. We named this entity proxy, because it acts as a proxy to the server. To comply with the near-far concept, it exhibits almost non existing latency towards that of the client, and performs invocations at the data center on the client's behalf.

This on its own will reduce the latency of the client towards the data center, which in turn will offload the network of the client. However, this doesn't change much performance-wise. Even if the proxy is to carry out the client's request, it would still ultimately have to be returned to that of the initial caller - which of course would result in the exact same time being spent.

To fully utilize the power of the near-cloud, one would have to provide the proxy with some functionality that would contribute to a reduction in its invocations to the data center. The incorporation of caching deems to address this very problem. Both the client and the proxy are to be equipped with caches, that gradually expands in size as the session goes on. Whenever a cache miss occurs at the proxy, not only will it fetch the data for the client, but it will also expand its own cache with a bigger chunk of data. This ensures that an increasing amount of future invocations will be targeted at the proxy rather than the data center - which theory should result in better performance.

To fully take advantage of our caching scheme, we are to combine a key concept that exists in the programming language Emerald, namely that of **attached** with the near-far cloud concept. The attached keyword works great with recursive data types, which is why we choose graph structures as our caches.

CHAPTER 5

DESIGN

This chapter serves as a blueprint for implementing the client-server and the near-far cloud architecture. The goal is to provide the reader with a fundamental understanding of the envisioned systems. BPMN diagrams [6] will be utilized in order to illustrate the inner workings of the caching algorithm associated with these two architectures.

5.1 Scope

Our goal is to demonstrate the feasibility of a graph-caching scheme that is based on the near-far cloud concept. We are to develop two prototypes in the Emerald programming language, and to compare them to one another by performing experiments on Planetlab.

5.2 Architecture

5.2.1 Data Structure

The underlying data structure that make up the caches are undirected graphs, consisting of one vertex root pointer that is recursively connected to other vertices by the use of the **attached** keyword of Emerald.

5.2.2 Algorithms

The key drivers of our system are the graph-search algorithms known as **depth-first search** and **breadth-first search**, as covered in our background chapter on graph theory. The depth-first search will be utilized for local cache look-ups as well as for the final evaluation algorithm. Breadth-first search will be used to expand vertices in our near-far deep caching.

5.3 Near-far cloud prototype

The system consists of three Planetlab nodes, where each node plays an associated role that corresponds to the near-far concept. The roles are as follows: client, proxy and data center. If there are numerous proxy nodes chained together, proxy node N+1 will be consulted upon a cache miss. If one has reached the last proxy node, and the data object was not found, the data center will be consulted. The data element is guaranteed to be located at the data center as it contains the full graph.

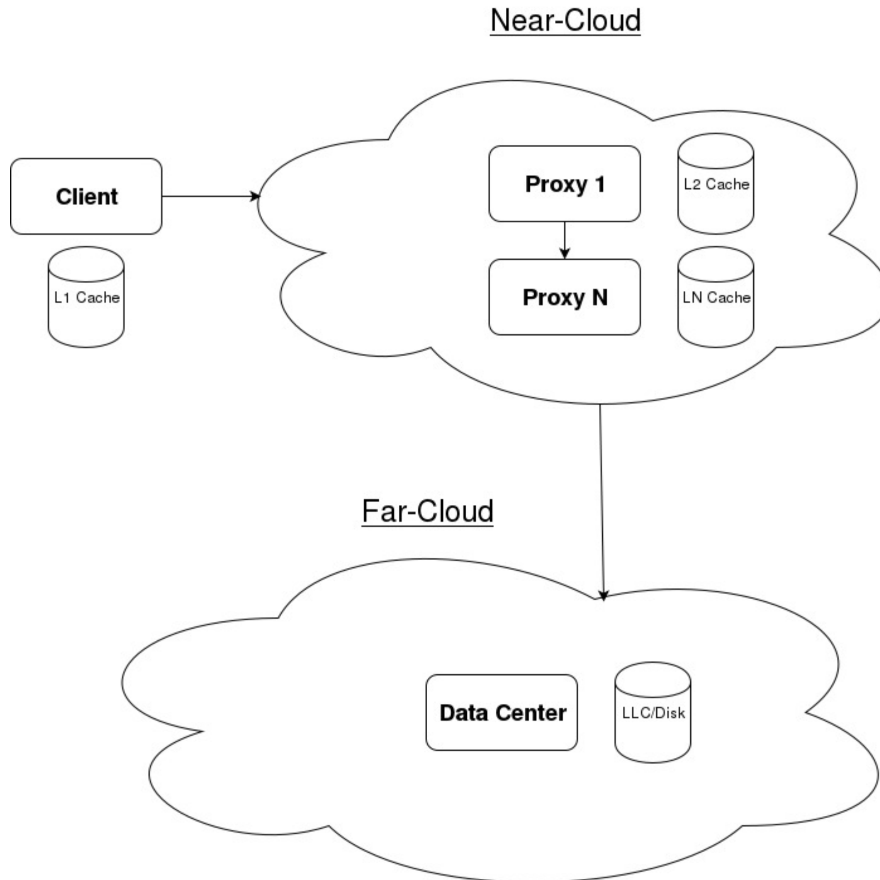


Figure 5.1: Illustration of the near-far cloud [24]

The client emulates a thin-client with limited resources. The thin-client is connected to the proxy, which acts as the near-cloud. As such, it provides the client with excellent latency and good computational power. The proxy is connected to the data center, which acts as the far-cloud. The far-cloud has superior storage capacity and computational power, but suffer poor latency towards the client.

Each role in the system is equipped with an in-memory cache of size that corresponds to their respective position in the cloud-hierarchy. The client has small-sized cache, the proxy a medium-sized cache and the data center a huge cache. Whenever the client is to request a data element, it starts by searching it's own local cache. If a cache-miss occurs, it will proceed to send a request to the proxy node for the data element. The proxy node will then search through it's own local cache for the data element.

5.3.1 Caching process near-far cloud

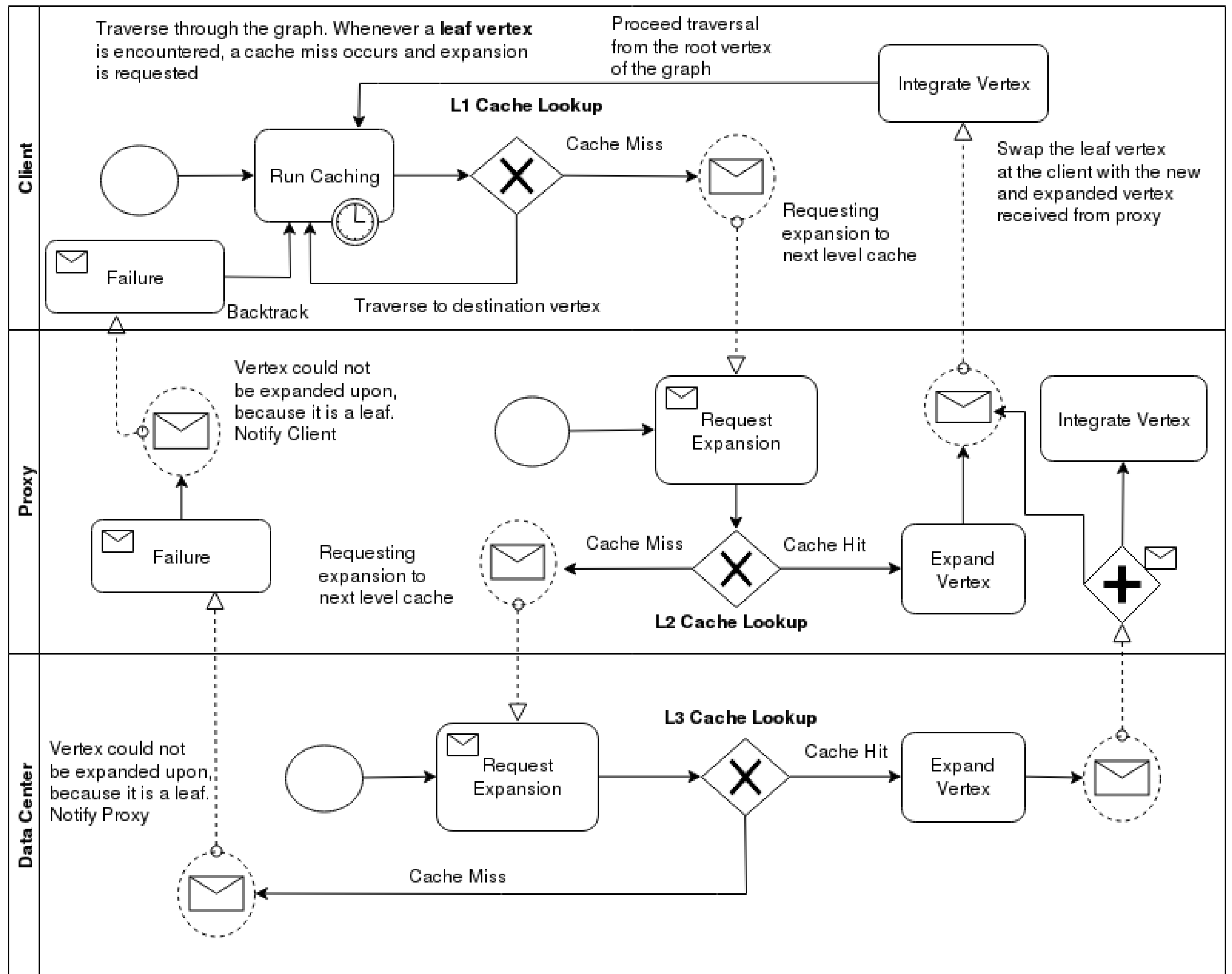


Figure 5.2: Near-far caching scheme [24]

As can be observed from our BPMN diagram in figure 6.11, there exists three *pools*; namely that of client, proxy and data center. These correspond to geographically distinct *nodes* that are deployed in Planetlab. The client is equipped with the first level cache (**L1**), the proxy the second level cache (**L2**) and the data center which is equipped with the third level cache (**L3**).

The algorithm starts at the client, where it will perform a continuous recursive depth-first search on its cache, which as mentioned is composed of a graph. This follows the same steps as a standard depth-first search as described in our section on Graph Theory. A target vertex will visit any neighbouring vertices through its edges that have not previously been visited. If the chosen vertex is a leaf vertex, it is to be expanded. Note that this implies that *cache hits are guaranteed to occur on vertices with outbound edges*, and at the same token, cache misses are guaranteed to occur on leaf vertices. This is an **important distinction** as cache hits and misses are usually based on the existence of a given element. Thus, the vertex with id "3" may be successfully found on the client's cache, but will still issue a cache miss - because it has no outbound edges (leaf vertex).

Upon the occurrence of a leaf vertex, one is first to perform a request for expansion to the proxy node. The proxy will perform a cache look-up on its local cache to see if the given vertex is eligible for expansion. If it is not a leaf vertex, a new vertex is constructed containing all its associated vertices. The newly expanded vertex will then get transmitted over to the client, and integrated with its cache. This mean that the old leaf vertex will be replaced with that of the newly expanded vertex. The recursive DFS will then restart from the parent of the newly expanded vertex (or the previous vertex prior to expansion).

However, if the target vertex was identified as a leaf at the proxy, it will continue to the data center node. The data center will search through its local cache in order to ascertain if the supplied vertex may be subject to expansion. If it indeed is eligible for expansion, it will construct a new vertex, and then ship it to our proxy node. At the proxy, it will integrate the vertex with its own cache, and then transmit the expanded vertex to the client. Finally, the client will integrate the vertex with its local cache. This is done so that the proxy is continuously growing, which will ensure that an increasing amount of invocations in the future will be targeted towards the proxy as opposed to the data center, which exhibits high latency towards both the proxy and that of the client.

If the data center was to identify a leaf vertex, which does happen, an error packet would be sent to the proxy, which would then continue to inform the client that the requested vertex could not be expanded. The client will then backtrack, and continue searching. Since the given vertex is marked as visited, it will not be subject to any more expansion requests.

5.4 Client-server prototype

The client-server prototype is based on the client-server architecture, and as the name implies, it is composed of two nodes; the client and the server. The server is a remote node that exhibits high latency towards the client. The client is equipped with a small cache and a cache algorithms that allows cache expansion. Since there are merely two nodes in this architecture, any cache miss on the client side results in a remote invocation to the server/data center.

5.4.1 Caching process client-server

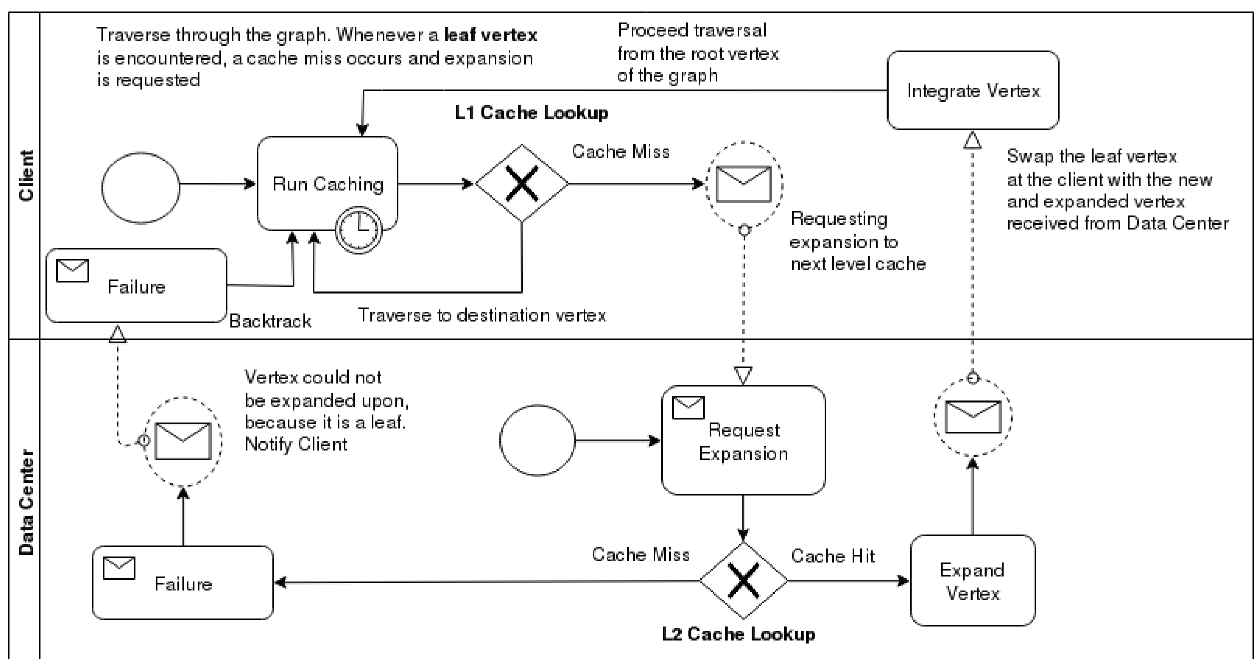


Figure 5.3: Client-server caching scheme [24]

The steps in our client-server caching scheme are strikingly similar to that of the near-far cloud caching scheme. The only real difference is that there is no intermediary cache, only the client and the data center

5.5 Summary

In this chapter we have outlined a plan for implementing our two architectures. The scope, architecture, data structure utilized and the process of caching have been examined.

CHAPTER 6

IMPLEMENTATION

This chapter is dedicated to the documentation of our prototype programmed in Emerald. Relevant object-constructors, typeobjects and algorithms will be outlined and briefly explained. If confusion is to arise, one is advised to consult the previous chapter on design as it offers better explanations of the system. This chapter is merely intended to illustrate the class structure and associated algorithms in Emerald.

6.1 Classes

The system to be implemented is comprised of eight core object constructors. For brevity we will refer to object constructors as classes, even though the notion of a class is merely syntactic sugaring for an object constructor.

The classes are to be divided into two groups: roles and data structure. The classes client, proxy and data center corresponds to the roles in the system that is described in section. They are subtypes of the abstract class Role, and as such conforms to it. The classes cache, graph, vertex and edge make up the data structure that is used as the cache. Cache has the possibility to create instances of objects that conform to the CacheAlgorithm typeobject, where DFSCaching is one of them.

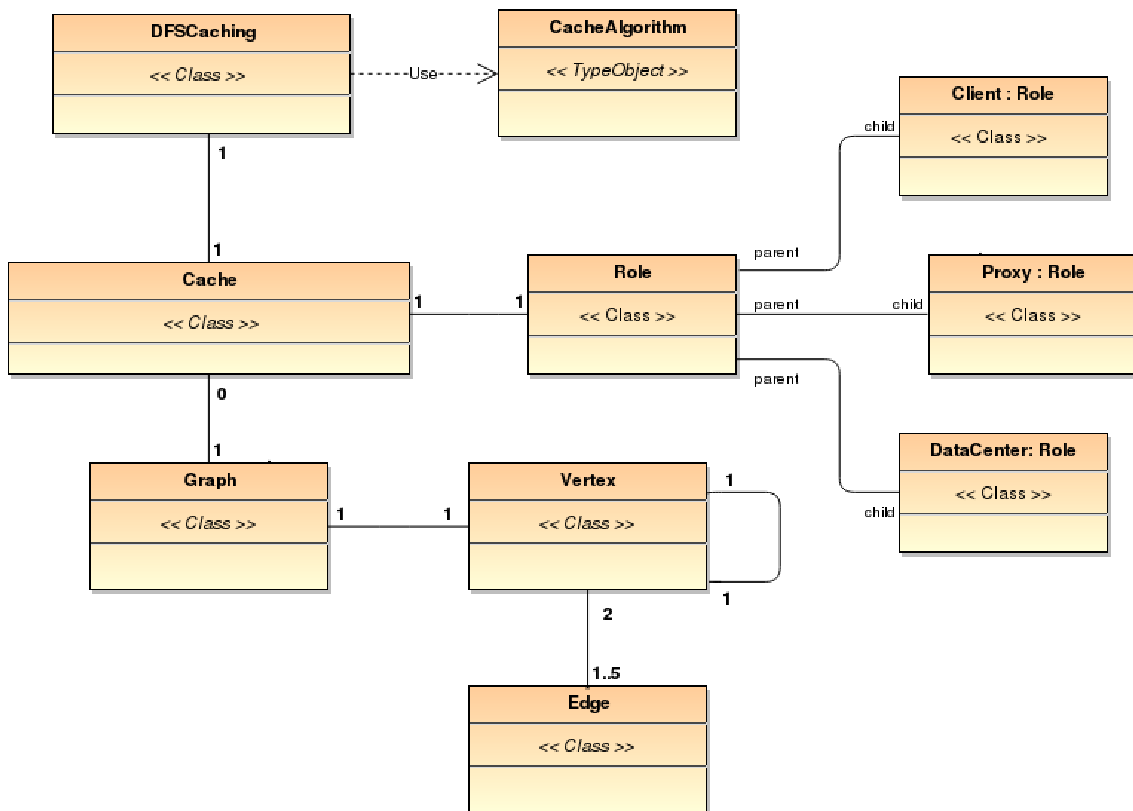


Figure 6.1: Class diagram outlining the components of the system

6.1.1 Roles

Class Role

The parent of the classes client, proxy and data center. It describes the set of operations that are deemed necessary in order to function as an active entity in our caching scheme. Any class that wish to act in the context of this class, has to conform to its signature functions and operations.

```
const Role <- Class Role

    export operation startRole
    export operation createCache
    export operation setNextLevelCache[nextLevel:Role]
    export function getCache -> [ret:Cache]
    export operation expandVertex[loc:Node, target:Vertex, mode:String] -> [ret:Vertex]

end Role
```

Figure 6.2: Functionality required in order to conform to type Role

Class Client

Client is responsible for the logic associated with the end-user of our system. It is supplied with a tiny cache, that is to expanded upon. Below is a summary of its attributes and functionality.

```
const Client <- Class Client(Role)

    var L1_cache:Cache

    export operation startRole
    export operation createCache
    export operation setNextLevelCache[nextLevel:Role]
    export function getCache -> [ret:Cache]
    export operation expandVertex[loc:Node, target:Vertex, mode:String] -> [ret:Vertex]

end Client
```

Figure 6.3: Attributes and functionality associated with Client

Class Proxy

The proxy acts as a mediator in between the client and that of the data center. It is to emulate the near-cloud in the near-far cloud concept. Similarly to that of the client, it has a cache. Below is a summary of its attributes and functionality.

```
const Proxy <- Class Proxy(Role)

    var L2_cache:Cache

    export operation startRole
    export operation createCache
    export operation setNextLevelCache[nextLevel:Role]
    export function getCache -> [ret:Cache]
    export operation expandVertex[loc:Node, target:Vertex, mode:String] -> [ret:Vertex]

end Proxy
```

Figure 6.4: Attributes and functionality associated with Proxy

Class Data Center

The data center is supposed to emulate the far-cloud in the near-far cloud concept. It is the most powerful role with respect to both computational power and storage capacity. It is equipped with a cache that contains all the vertices in the system. Thus, cache hits are guaranteed at the data center. However, since it resides far away from the proxy, calls to it are naturally expensive.

```
const DataCenter <- Class DataCenter(Role)

    var L3_cache:Cache

    export operation startRole
    export operation createCache
    export operation setNextLevelCache[nextLevel:Role]
    export function getCache -> [ret:Cache]
    export operation expandVertex[loc:Node, target:Vertex, mode:String] -> [ret:Vertex]

end DataCenter
```

Figure 6.5: Attributes and functionality associated with DataCenter

6.1.2 Data structure

Class Cache

The cache is comprised of a graph, and up to several different caching algorithms that may be activated. This is to support loose coupling in the sense that it doesn't matter which algorithm that is used, as long as it provides the required operations which are specified in the typeobject. This is also a good way of providing abstraction.

```
const Cache <- Class Cache

    var data:Graph
    var algorithm:CacheAlgorithm
    var nextLevelCache:Role

    export operation cache_executeCaching
    export operation cache_createCache[size:Integer]

end Cache
```

Typeobject CacheAlgorithm

This typeobject acts as a template that our caching algorithms has to conform to. This ensures that the system is loosely coupled in the sense that one may provide the cache with any caching algorithm as long as it operations is a superset that of the typeobject CacheAlgorithm.

```
const CacheAlgorithm <- Typeobject CacheAlgorithm

    operation cacheData[root:Vertex]
    operation setCache[target:Cache]

end CacheAlgorithm
```

Figure 6.6: Functionality required in order to conform to type CachingAlgorithm

Class DFSCaching

This class is the main cachingalgorithm that will be utilized in this thesis. As the name implies, it is based on depth-first search. Our caching algorithm is to expand leaf vertices, and as such the DFS was the natural choice of graph algorithm.

```
const DFSCaching <- Class DFSCaching

    var associatedCache:Cache

    export operation cacheData[root:Vertex]
    export operation setCache[target:Cache]

end DFSCaching
```

Figure 6.7: Attributes and functionality associated with DFSCaching

Class Graph

Every cache has a graph of a given size that may be expanded upon through caching. The graph contains a single pointer to a vertex that further points to other vertices through the utilization of the attached concept. Graph algorithms are using the root pointer to traverse the graph recursively, rather than by the use of adjacency lists or adjacency matrices.

```
const Graph <- Class Graph

    var size:Integer
    var root:Vertex

    export operation graph_recursiveDFS[root:Vertex, target:Integer] -> [ret:Vertex]
    export operation graph_expandVertex[target:Vertex, mode:String] -> [ret:Vertex]
    export operation graph_generateGraph[initialSize:Integer] -> [ret:Graph]

end Graph
```

Figure 6.8: Attributes and functionality associated with Graph

Class Vertex

Vertices are the building blocks that make up graphs. Each vertex has up to five edges that are attached to other vertices. This relation works recursively. It is this utilization of recursive data types that enables the use of the graph-caching.

```
const Vertex <- Class Vertex

    attached var id:Integer
    attached var parent:Vertex
    attached var edges:List.of[Edge]

end Vertex
```

Figure 6.9: Attributes and functionality associated with Vertex

Class edge

Edges acts as bridges to other vertices. Similar to the vertices, the attached keyword is used here.

```
const Edge <- Class Edge

    attached var source:Vertex
    attached var destination:Vertex

end Edge
```

Figure 6.10: Attributes and functionality associated with Edge

6.2 Key Algorithms

In this section we will outline the most essential algorithms utilized in our system; "cacheData" and "partialExpansion". The operation "expandVertex" has been included despite its crude nature because it is being used in our examples.

6.2.1 cacheData

```
cacheData(currentVertex)

        if cache.size = GOAL then

                  exit

        else if currentVertex.getAmountEdges < 1 then

                  do cache ← nextLevelCache.expand(currentVertex)

                  if expansion successful
                      do cacheData(currentVertex.getParent)
                  else
                      return
                  end if
        else

                 for each children ∈ currentVertex.getEdges

                      if children ≠ visited then
                         do children ← children.visit
                            cacheData(children)
                      end if
        end if
```

Figure 6.11: cacheData

6.2.2 expandVertex

```
expandVertex(targetVertex, mode)

        if mode = "PARTIAL" then
           do partialExpansion(targetVertex)
        else if mode = "DEEP" then
           do deepExpansion(targetVertex)
        end if
```

Figure 6.12: expandVertex

6.2.3 partialExpansion

```
partialExpansion(targetVertex) -> (returnedVertex)

      expandedVertex ← Vertex.create(targetVertex.getID)
      edges ← targetVertex.getEdges

      for each edge ∈ edges
          do destination ← edge.getDestination
             newVertex ← Vertex.create(destination.getID)
             newVertex ← newVertex.setParent(expandedVertex)
             newEdge ← Edge.create[expandedVertex, newVertex]
             expandedVertex.addEdge[newVertex]
      end for

      returnedVertex ← expandedVertex
```

Figure 6.13: expandVertex

6.3 Summary

In this chapter we explained our implementation. Core classes, algorithms and interactions have been highlighted and illuminated. The reader is to understand the core constituents that make up the architecture and the algorithms utilized.

CHAPTER 7

EVALUATION

In this chapter we are to present our findings. Experiments have been carried out in order to establish baseline performance measures for both local and distributed algorithms. By having performance metrics for local algorithms, one may better understand the distributed ones, especially when there are evident bottlenecks that otherwise would just naively be accepted "as part of the algorithm".

Performance of local algorithms will be performed at the relevant nodes; which is the nodes associated with the roles proxy and data center. This is done to reduce the risk of confounding variables.

7.1 Memory limitations in emerald

Emerald comes with a restriction on the amount of memory that may be in use at any given time. This seems to be an issue that at least now cannot be worked around. Figure 7.1 provides a screen-shot of the error that occurs when memory do run out. The "flag" that is being mentioned does not work.

```
old_end = 0xf7403008, next_gen = 0xf7395c9c, diff = 111835, promo = 0

Out of memory.  Try including the flag -O2304k on the command line!
```

Figure 7.1: Out of memory prompt

We have conducted two experiments to demonstrate when memory runs out; one by using arrays, and other using graphs. The results of the former is illustrated in figure 7.2, while the latter in figure 7.3.

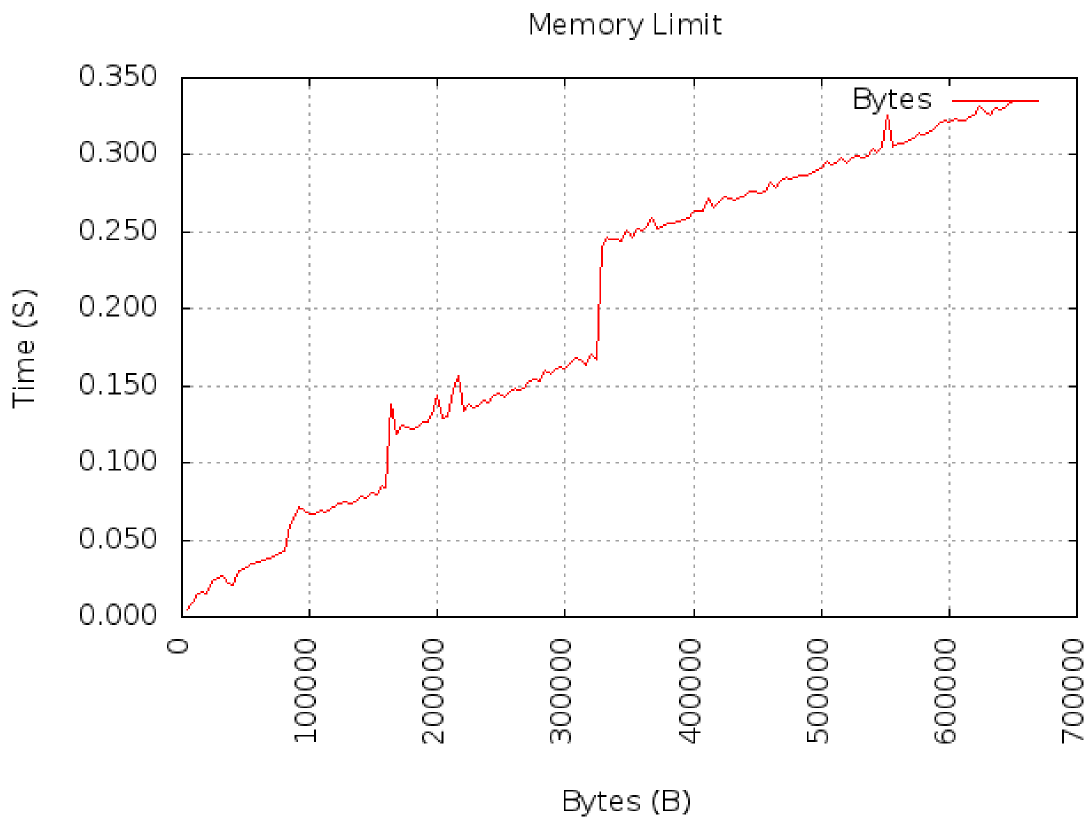


Figure 7.2: Memory runs out at approximately 652000 bytes

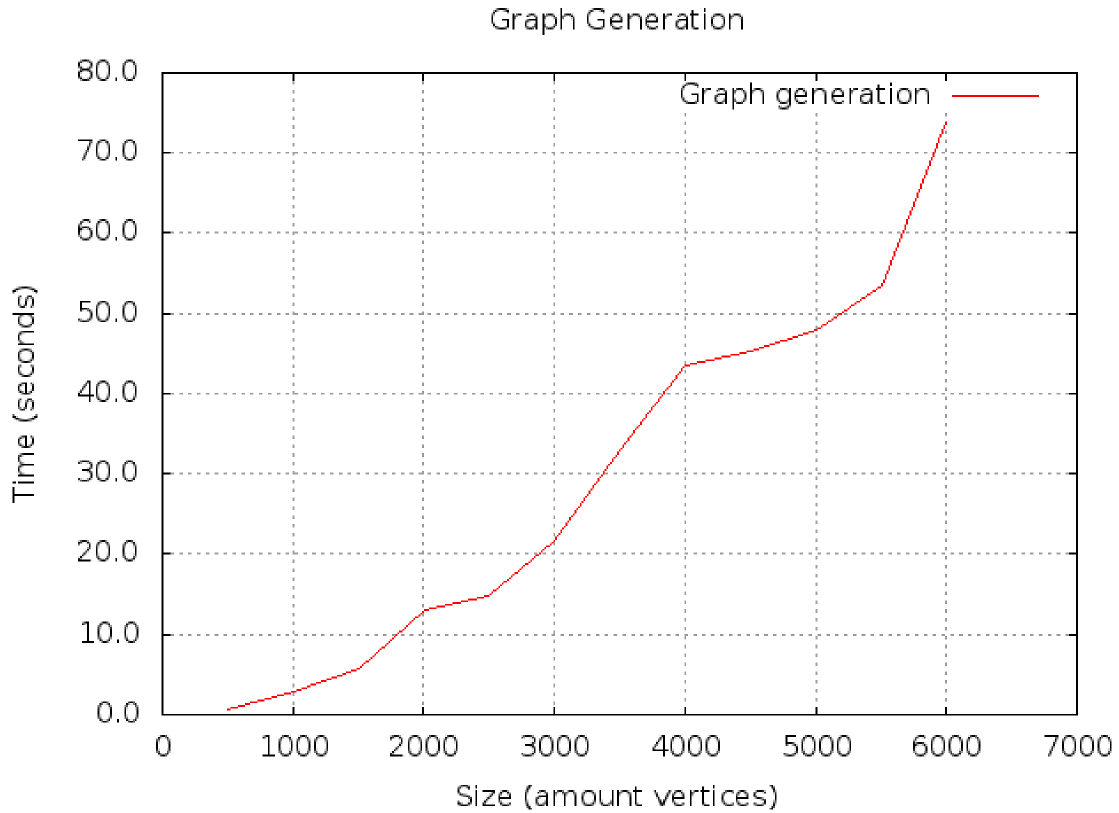


Figure 7.3: Time taken to generate graph

7.1.1 Implications of memory constraints

The implications of this constraint is that one cannot generate graphs beyond 6000 vertices, which is why we operate with graphs in the 5-6000 range at maximum. This is not really a big deal, since a graph of size 5000 is more than enough to demonstrate our system. Our original plan was to associate every vertex with a specific amount of *padded bytes*. This way we could emulate load, and thus have a bigger impact on data transmission. When data objects are too small, the latency will *overshadow* the overhead associated with data size. This will be demonstrated in the coming pages.

7.2 Performance metrics

7.2.1 Nodes and latency

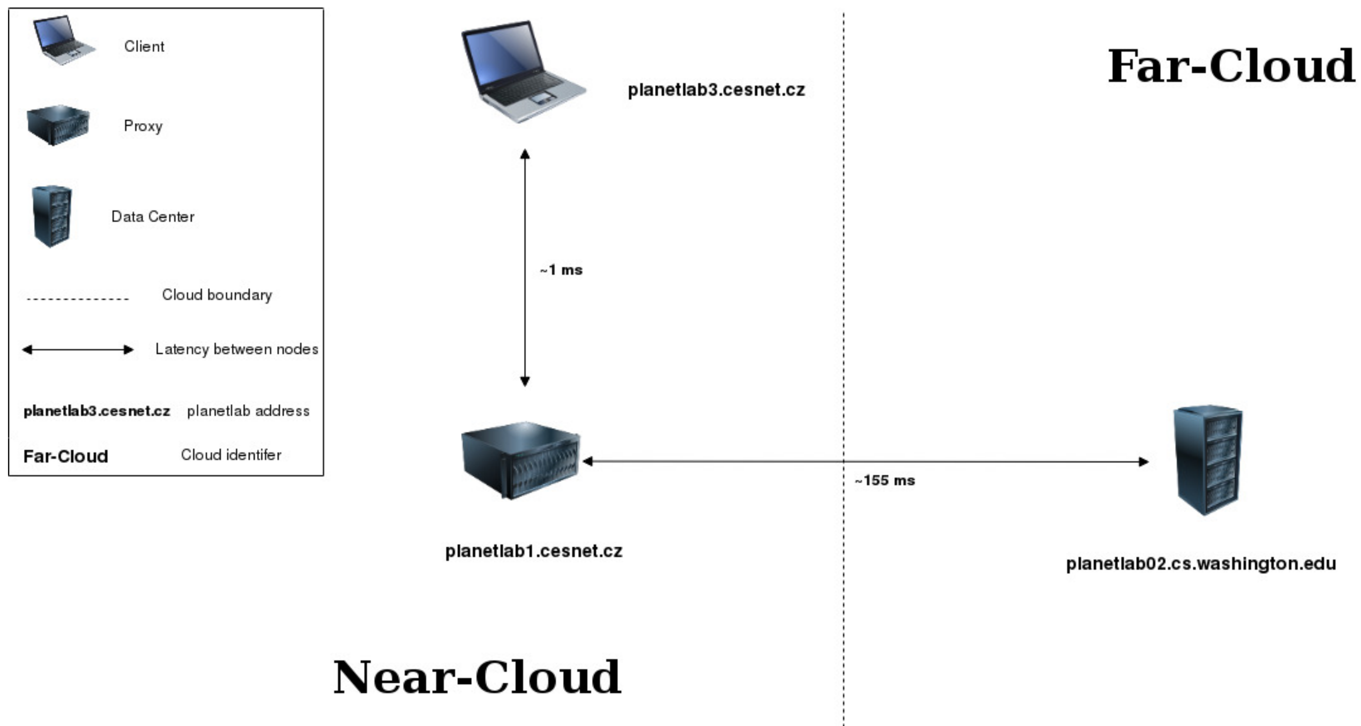


Figure 7.4: Illustration of the Planetlab nodes [24]

```
--- planetlab02.cs.washington.edu ping statistics ---
380 packets transmitted, 379 received, 0% packet loss, time 379141ms
rtt min/avg/max/mdev = 154.796/154.968/160.914/0.476 ms
[diku_turtle@planetlab3 ~]$
```

Figure 7.5: Ping results towards data center

```
--- planetlab1.cesnet.cz ping statistics ---
370 packets transmitted, 370 received, 0% packet loss, time 369015ms
rtt min/avg/max/mdev = 0.095/0.146/0.575/0.045 ms
[diku_turtle@planetlab3 ~]$
```

Figure 7.6: Ping results towards proxy

7.2.2 Performance of local algorithms

In order to identify potential bottlenecks, and to ascertain if there exists any differences with regards to computational power between the various nodes, key local algorithms are to be benchmarked.

We will cover an extensive analysis of our depth-first search as it is essential to this thesis.

Generate graph

Figure 7.7 provides an illustration of performance associated with graph generation. This experiment was performed on our node associated with the proxy role; *planetlab1.cesnet.cz*.

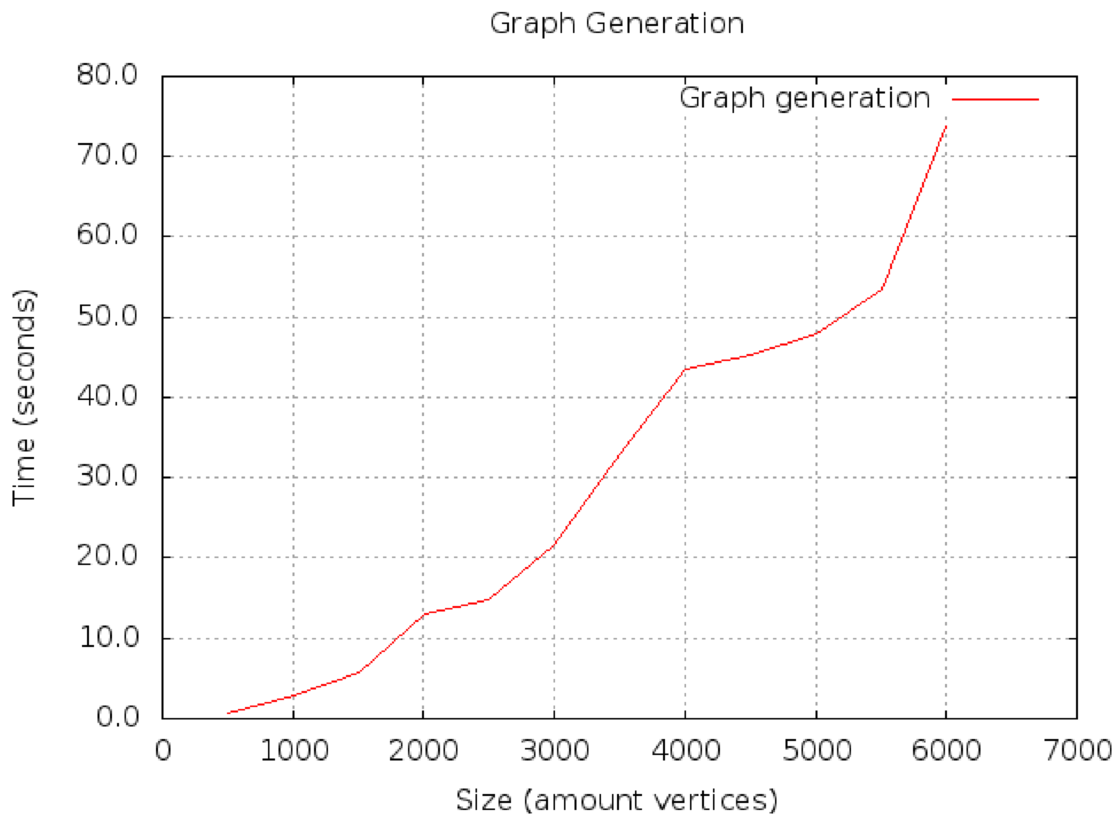


Figure 7.7: Graph generation performance

Transmit graph

Figure 7.8: Graph transmission overheads

As previously demonstrated, the data center node exhibits a latency of 155 ms towards the proxy node. It is evident from figure 7.8 that the majority of transmission time is due to this inherent latency. Data size does not seem to have much of an impact, because the relative size of objects in bytes are small. There is little difference in transmitting a graph of size 1000 compared to one of 6000. Thus, the transmission time overhead associated with a graph consisting of 5 vertices is essentially the same as the overhead associated with a graph consisting of 100 vertices. The data elements themselves are not big enough to have any real impact on the total performance, in comparison to the speed of light that is.

Recursive DFS

In this subsection we are to investigate the performance metrics associated with the depth-first search, which is a recursive graph search algorithm. Depth-first search is used for cache look-ups as well as being the basis for our expansive depth-first search that is covered later in the chapter. We are performing 5000 invocations of worst-case DFS. Worst-case refers to searches that are guaranteed to traverse every vertex in the graph. We are then calculating the average to establish a baseline. We do this because it is our belief that a pessimistic approach to evaluation will produce more realistic results.

These experiments will be performed on graphs of size 1000 and 6000. The nodes associated with our roles *proxy* and *data center* will be the subjects of our study. The reason we do this is to investigate to which degree size influences cache-look speed, but also whether or not there exists performance discrepancies between the Planetlab nodes.

The nodes of study are "planetlab02.cs.washington.edu", which plays the role of data center, and "planetlab1.cesnet.cz", which plays the role of proxy.

Proxy (Czech Republic)

Figure 7.9 and 7.10 illustrates the worst-case performance associated with DFS on graphs of size 1000 and 6000. The average search time at size 1000 was *15.2 ms*, while at size 6000 the average was *106.2 ms*.

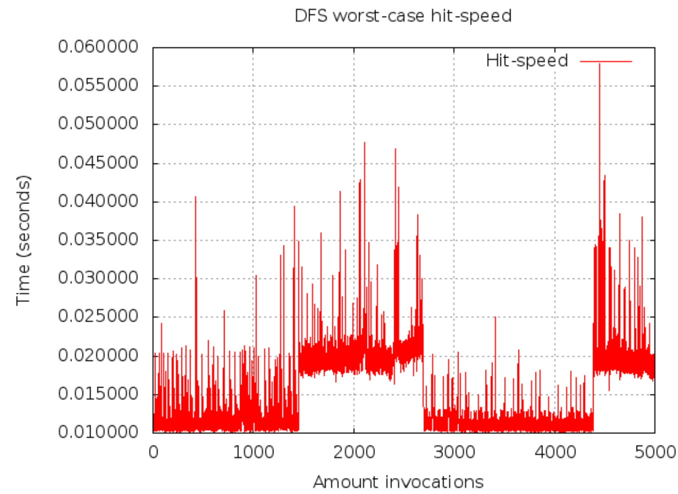


Figure 7.9: DFS worst-case at size 1000

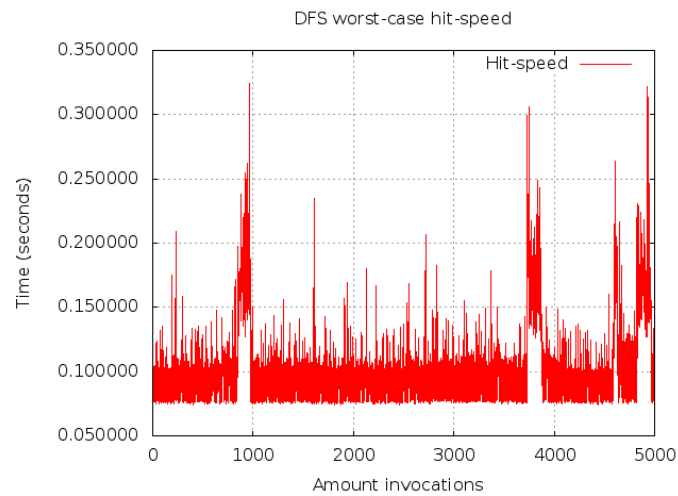


Figure 7.10: DFS worst-case at size 6000

Data Center (United States)

Figure 7.11 and 7.12 illustrates the worst-case performance associated with DFS on graphs of size 1000 and 6000. The average search time at size 1000 was *40.6 ms*, while at size 6000 the average was *343.2 ms*.

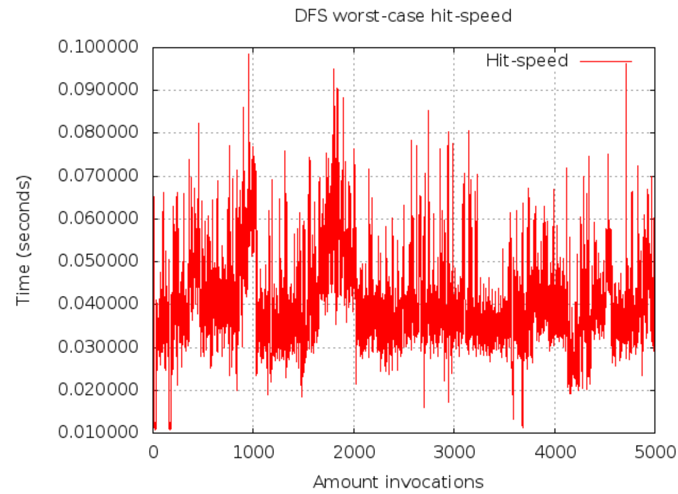


Figure 7.11: DFS worst-case at size 1000

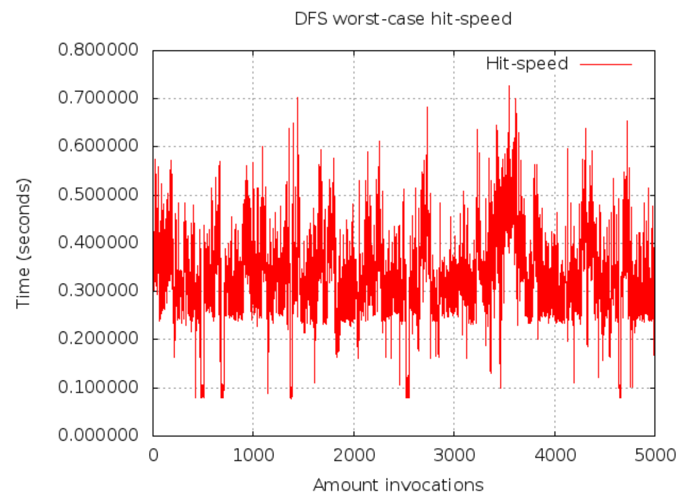


Figure 7.12: DFS worst-case at size 6000

Comparison between nodes

We identified a *significant* discrepancy in performance between the nodes in terms of average search overhead. At graph size 6000, the data center had an average worst-case search speed of 343.2 ms, whereas the proxy had an average of 106.2 ms. This amounts to a difference in latency between the nodes of 237 ms, which is quite large. We agreed that this is not acceptable, as it exceeds the latency towards the data center by far. It also contradicts the near-far cloud concept, where the data center is supposed to exhibit greater computational power than that of the proxy.

New Data Center Node

We decided to search for another Planetlab node with more fitting performance capabilities. After benchmarking various nodes, we identified the perfect candidates. A node based in Canada that goes by the hostname of "planetlab1.cs.ubc.ca" was deemed perfect for our experiments. Its latency towards our existing proxy node averaged at 158.29 ms and exhibited an average worst-case search time for DFS at size 6000 at 65.46 ms. This is a significant reduction from that of the old data center, and also a noticeable reduction from that of our proxy node. As previously shown, the proxy node exhibited average worst-case look-ups of 106.2 ms. Now that our data center node has improved computational power in comparison to that of the proxy node, it neatly fits in with the near-far concept. To recap, nodes that are residents into the "far-cloud" are to have superior storage capacity and improved computational power in comparison to the services located at the near-cloud. Figure 7.13 provides a screen-shot of latency measures from our proxy node towards the new data center, located in Canada.

```
 planetlab1.cs.ubc.ca ping statistics ---
 packets transmitted, 301 received, 0% packet loss, time 300389ms
 min/avg/max/mdev = 157.229/158.296/162.317/0.722 ms
ku_turtle@planetlab1 ~]$
```

Figure 7.13: Latency measures

Data Center (Canada)

Figure 7.14 illustrates the average worst-case performance associated with a DFS search on a graph of 6000 vertices. Total experiment time amounted to 327.3 seconds. The average search time was 0.06546 seconds, which corresponds to *65.46 ms*. This is a huge improvement from the previous US based data center, and should aid in providing more realistic results.

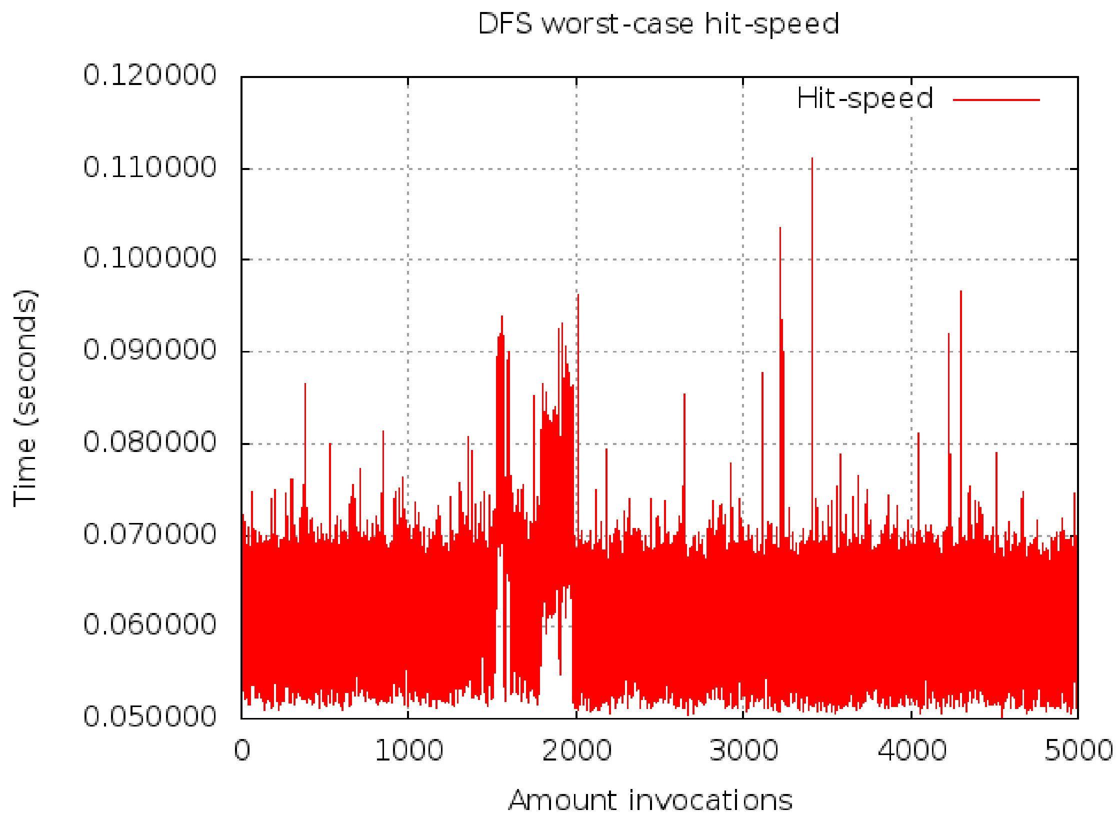


Figure 7.14: DFS worst-case at size 6000

7.2.3 Caching

We are now to demonstrate our caching scheme, by comparing performance associated with the near-far caching scheme to that of the client-server caching scheme. The near-far caching scheme consists of three nodes that make up our multi-level cache: client, proxy and data center. Note that our original node associated with the data center role "planetlab02.cs.washington.edu" has been replaced with the node "planetlab1.cs.ubc.ca". The client-server architecture consists of two nodes: client and server. We are to compare client-server *partial caching* to that of near-far cloud partial caching, and near-far cloud partial caching to that of near-far cloud *deep caching*. A custom depth-first search, named *expansive depth-first search* will be employed to demonstrate the different caching schemes. To demonstrate the impact of latency, we start by comparing performance between client-server partial caching of a high latency server with that of client-server partial caching of a minuscule latency server. This is done to illustrate the impact of latency when packet-exchange is frequent.

Expansive DFS

Expansive depth-first search works similar to a standard recursive DFS, with the difference being that whenever a leaf vertex is encountered, it attempts to *expand* the given vertex by caching it. This is done by consulting the next level cache. If the target leaf vertex was successfully expanded, the algorithm will restart from the parent of the given vertex (the target vertex will be unmarked before the traversal). This way it keeps expanding into the depths. It is a brute-force algorithm in that it will attempt to query any vertices for potential expansion, even though they have been visited before (as the surrounding area gets unmarked in the aftermath of a recent expansion). This is done entirely on purpose to ensure a high frequency of invocations towards the next level cache.

Caching performance comparison

We are to benchmark both total time spent accumulating caches, and the amount of data center/proxy invocations that were required in accumulating caches. For the client-server scheme, we will adopt a client cache size of 50 and a proxy cache size of 5000. For the near-far scheme, we will experiment with various initial cache sizes as the *deep caching* algorithm is affected by this.

Client-server caching (minimal latency)

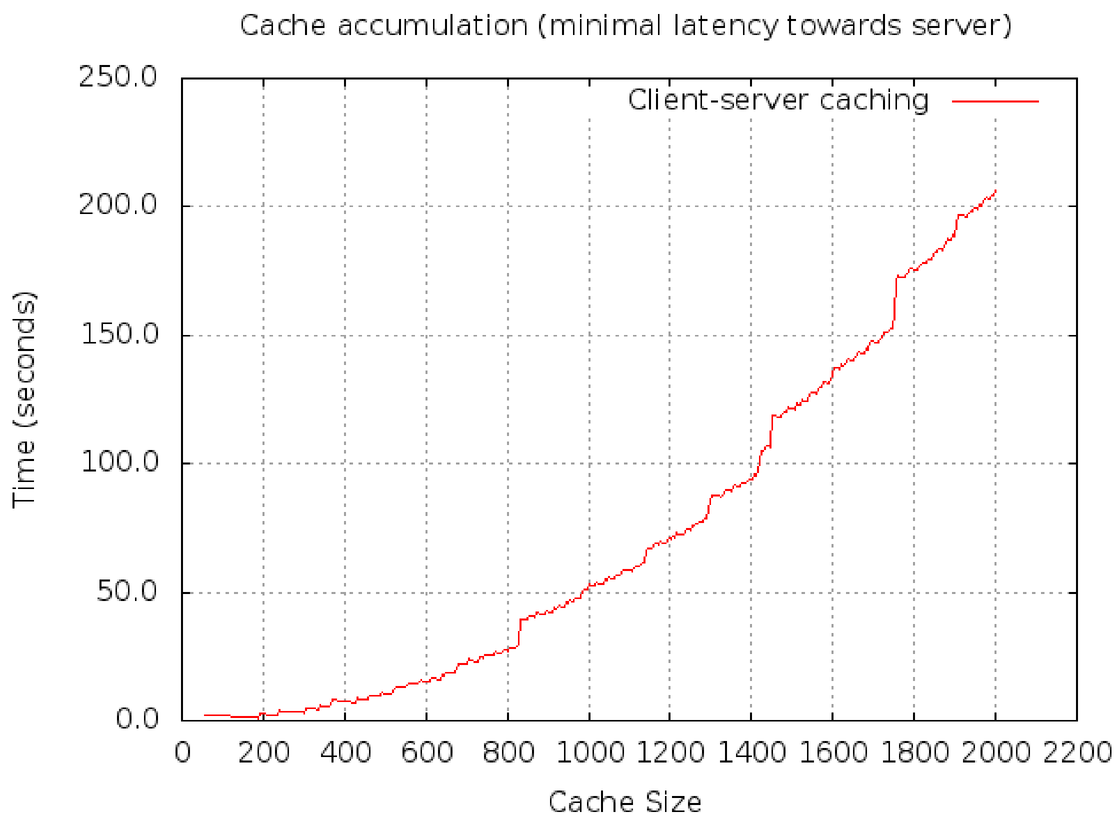


Figure 7.15: Client-server caching with server of minimal latency

The client exhibits a latency of 0.146 ms towards the server. The relationship between cache size and time spent may be observed in figure 7.17.

Client-server caching (high latency)

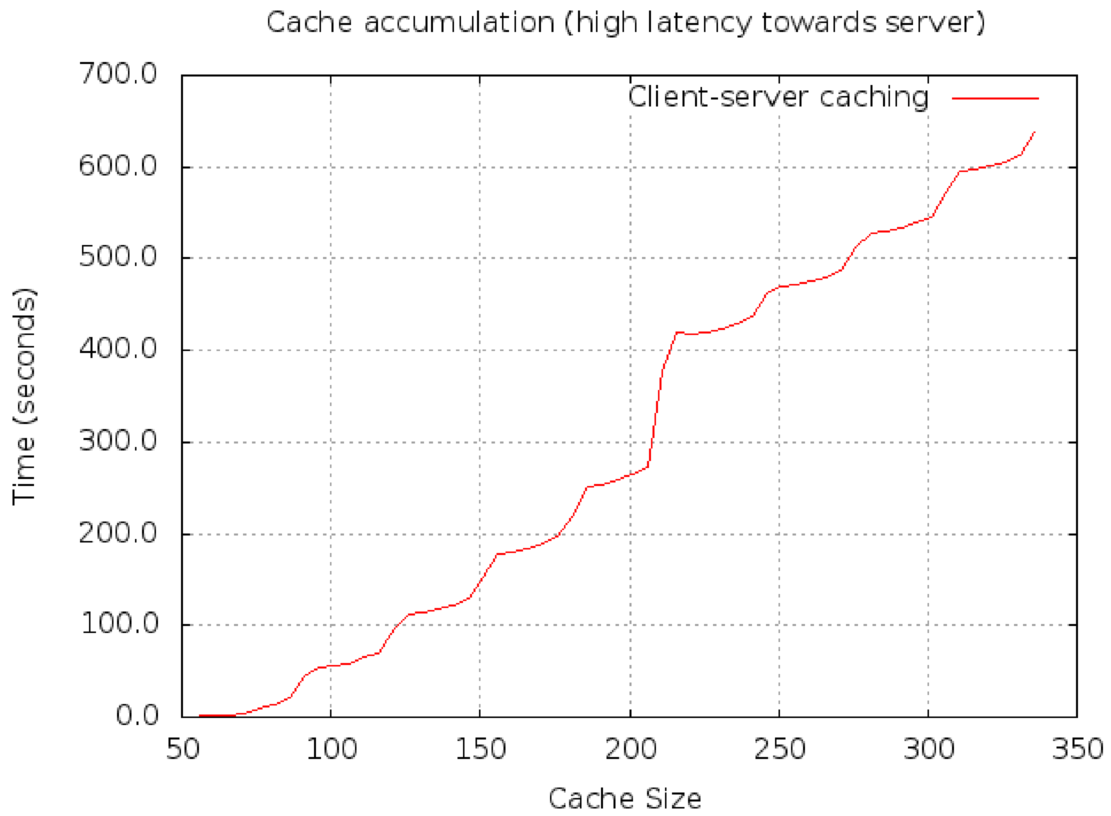


Figure 7.16: Client-server caching with server of high latency

When we upped the ante, and tested on a server of considerably higher latency (158 ms), performance degraded severely. We had to stop the experiment at size 330, as it had already passed the 600 second mark.

Remote invocations

To understand why there exists a huge performance discrepancy between a "no latency" and "high latency" remote node, one has to look the amount of remote invocations spent. As previously discussed, every remote invocation/packet transmission incurs an overhead that corresponds to the latency that exists between the nodes.

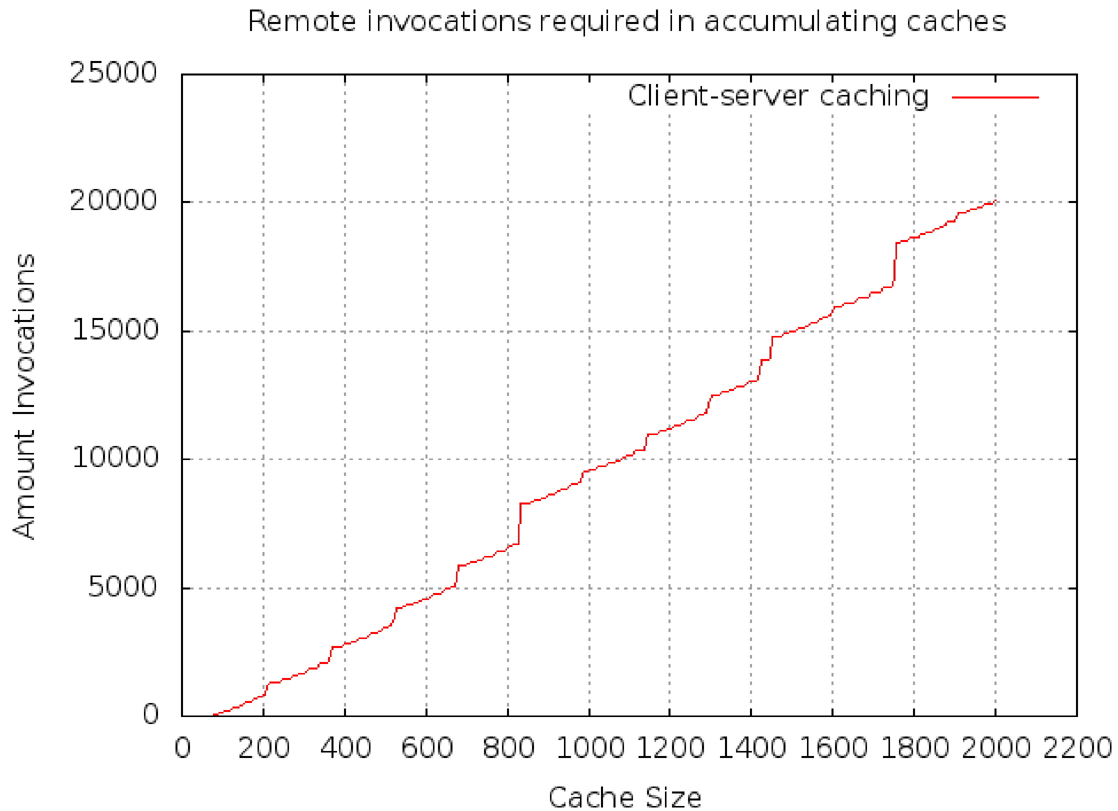


Figure 7.17: Invocations associated with client-server caching

As is evident from figure 7.19, there is a lot of remote invocations going on. At cache size 300, there has been somewhere between 1000 and 1500 remote invocations. Since we have demonstrated that our new data center is actually *faster* than that of the proxy, it is safe to conclude that it is not the cache-look-ups that are to blame. Thus, the performance impairment stems from the added latency, and not because of increased cost of cache-look-ups (which ironically has become cheaper).

Near-far cloud caching

The near-far scheme is multi-level caching scheme with an added intermediary role, known as proxy. We are to experiment with different sizes for the client, proxy and data center. The various size will be annotated as such "[25|250|5000]", which corresponds to a client cache of size 25, proxy cache of 250 and data center cache of size 5000.

While the client-server is guaranteed to incur an overhead that corresponds to the latency in case of a local cache-miss, in the near-far cloud scheme the proxy is consulted instead. We are to first look at the partial caching, and then look at the more advanced *deep caching*

Partial caching

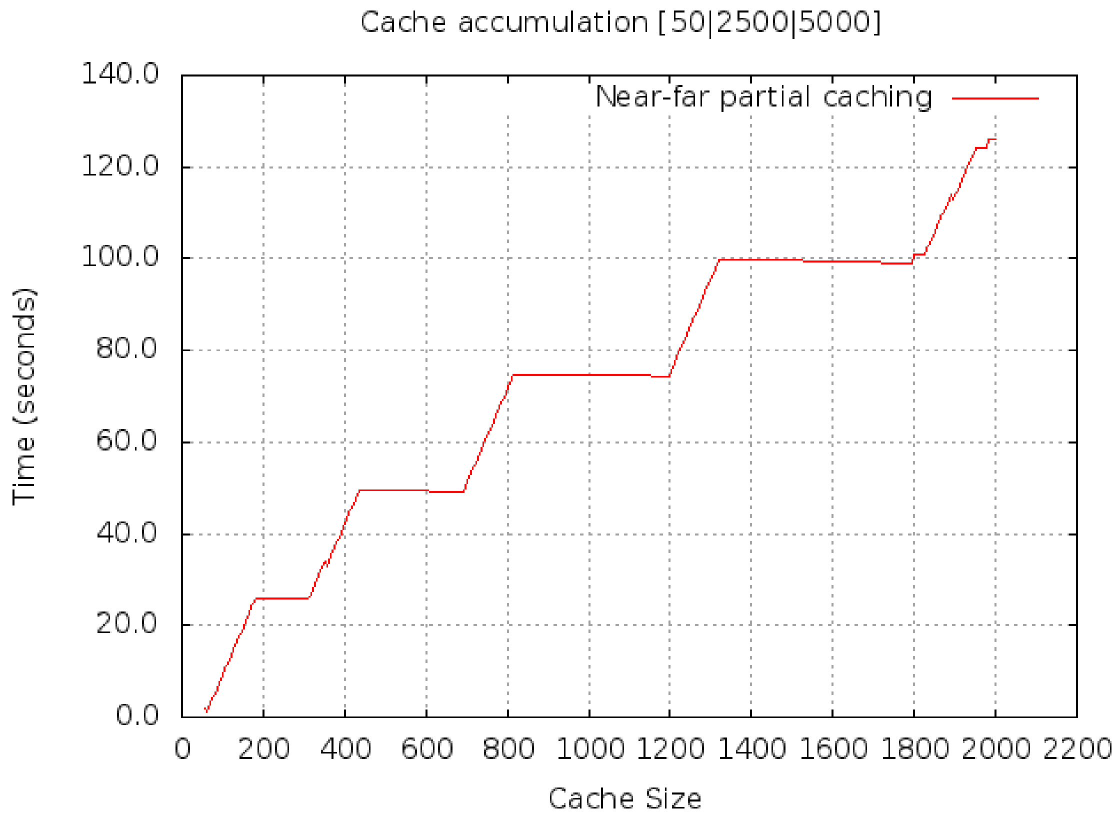


Figure 7.18: Near-far partial caching [50|2500|5000]

Client is equipped with a cache size is 50, proxy a cache of 2500 and data center cache of 5000. The scheme resulted in 390 proxy invocations and 125 data center invocations. The total elapsed time was 125 seconds in accumulating a cache of size 2000.

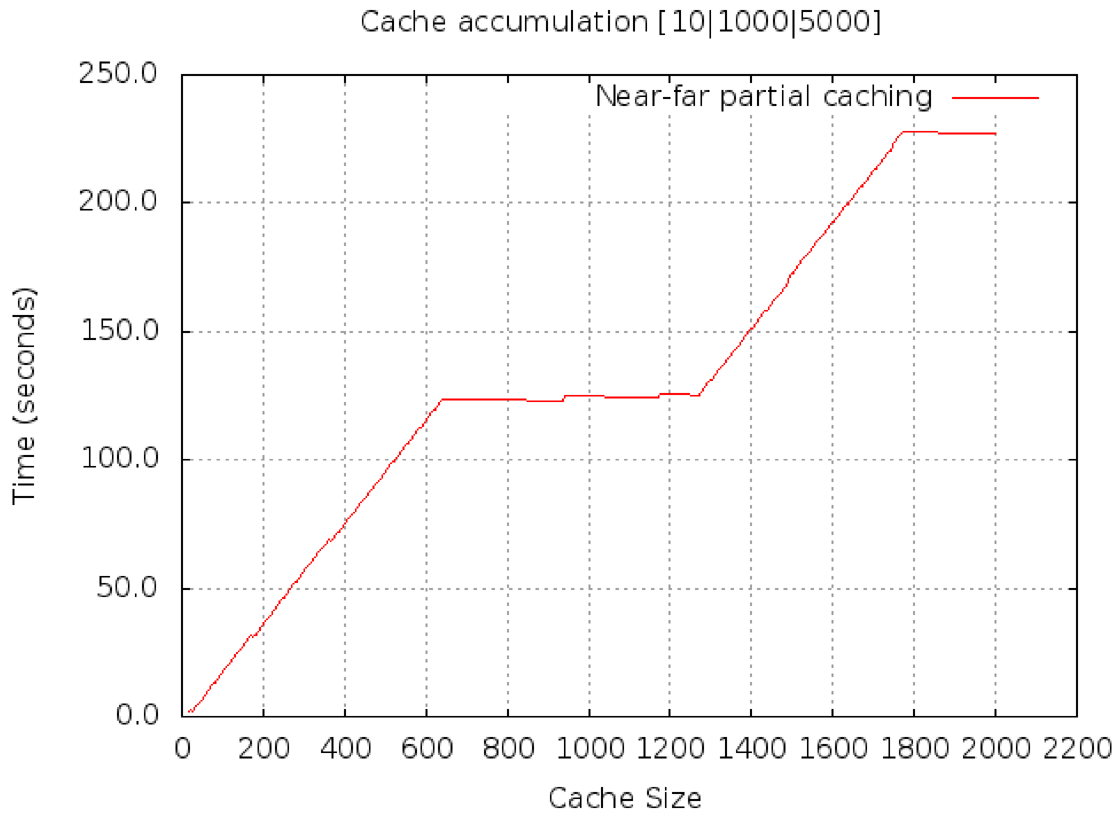


Figure 7.19: Near-far partial caching [10|1000|5000]

Client is equipped with a cache size is 10, proxy a cache of 1000 and data center cache of 5000. The scheme resulted in 399 proxy invocations and 226 data center invocations. The total elapsed time was 227 seconds in accumulating a cache of size 2000.

Deep caching

This more experimental mode of caching attempts to fully expand a given vertex, as opposed to merely expand it by 1 depth worth of vertices. It will recursively append descendant vertices onto the target vertex. This is a more risky approach as we experienced several times that the client cache would outgrown that of the proxy cache, which would result in it having vertices that the proxy didn't. This contradicts the whole inclusive caching scheme. This may be circumvented by a more thorough implementation, but we didn't have the time to revamp the scheme as complexity had already risen to a high level.

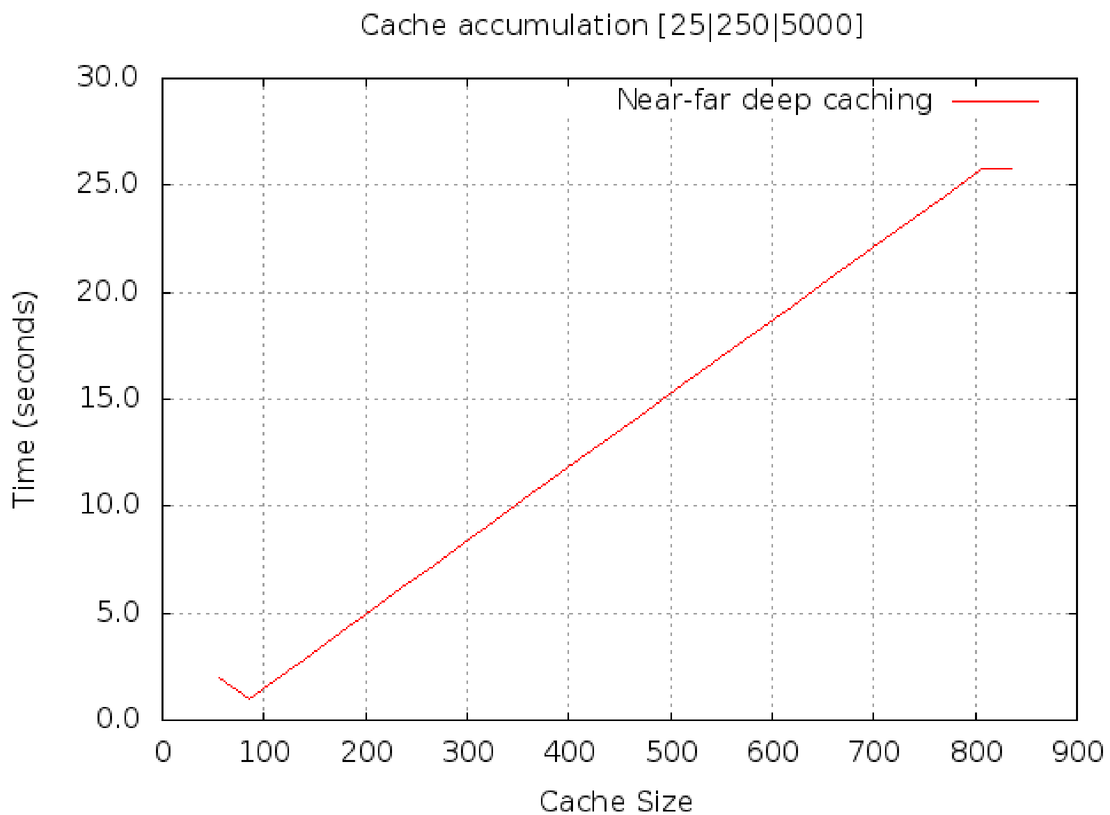


Figure 7.20: Near-far deep caching [25|250|5000]

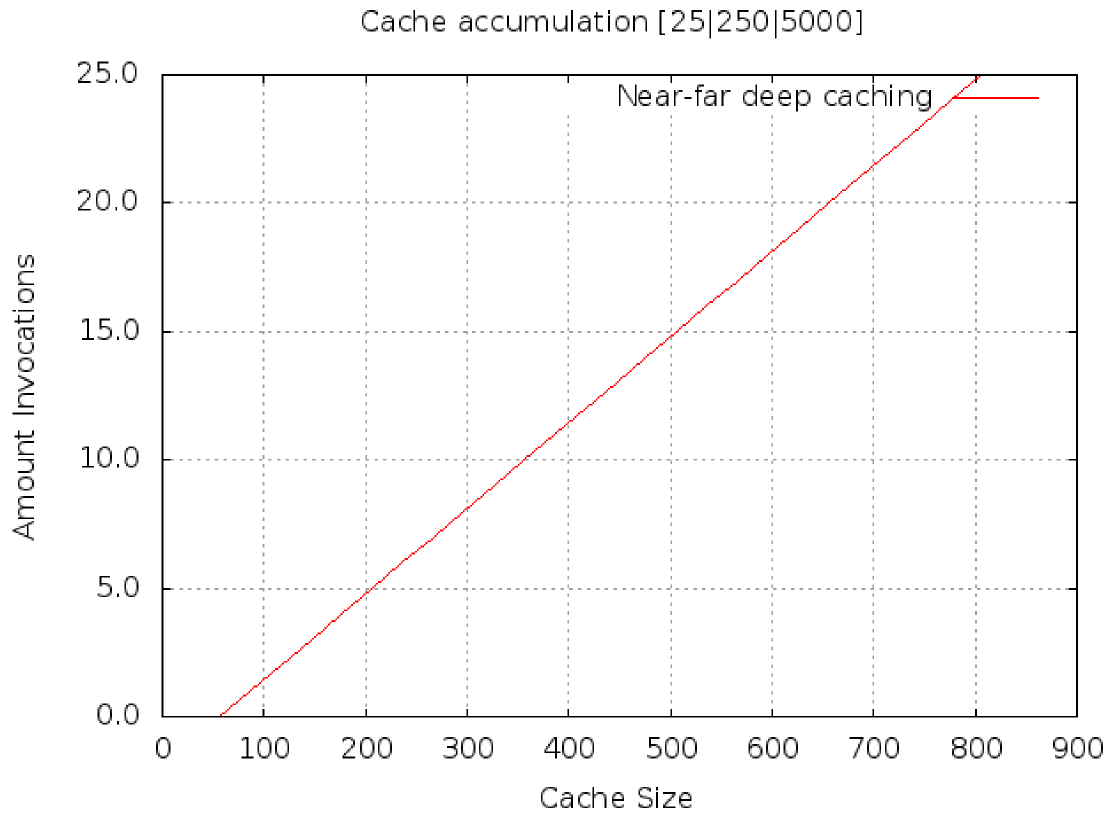


Figure 7.21: Near-far deep caching [25|250|5000]

7.3 Analysis

7.3.1 Findings

From our experiments we can conclude that frequent invocations towards high latency remote nodes are indeed expensive. Our comparison of the client-server caching scheme with varying latency of the remote node illustrates this clearly. When performing the caching on a remote node of close to non-existing latency (0.1 ms), the performance is stable. However, the moment one adds latency (158 ms) into the equation, the performance degrades profoundly. This is because *Over a thousand invocations are being made to the remote node*. This is not to say that the latency itself is the only factor that influences the performance drop, but it is definitely a big part of it.

The near-far partial caching scheme provides a significant performance increase

by reducing the amount of invocations to the data center, which exhibit a latency of 158 ms towards the client. Deep caching provides impressive performance increase, but comes with the added caveat that there exists issues with the algorithm that have to be fixed for it be at its full glory.

Cache look-ups may be potentially as devastating as latency. At bigger cache sizes (5-6000), the associated overhead of cache look-ups is significant. The overhead heavily depends on the node in question, as have been indicated. It is evident that on Planetlab, not all nodes area created equal as far as computational power goes. There exists severe performance discrepancies. This is something that one should keep in mind when performing experiments.

7.3.2 Scalability

Emerald comes with an implicit lack of structural scalability as there exist a maximum capacity on memory. Both our prototypes exhibits decent space scalability. This is because the graph caching minimizes the amount of vertices being in use by taking advantage of recursive data types rather than utilizing adjacency lists. However, the incorporation of this type of graph structure also leads to poor load scalability because of its algorithmic complexity.

7.4 Source Code

All code associated with our thesis is to be found at the following git repository that is being cited [1]. A thorough guide on how to perform our experiments will be provided on this repo.

7.5 Summary

In this chapter we have presented our evaluation of the system. Experiments have been conducted to benchmark performance of local algorithms, as well as distributed ones. Cache size was identified as a major influence on performance as cache size grows. Performance discrepancies among Planetlab nodes were also identified. The-

near far caching scheme was proven superior to that of the client-server caching scheme for caches of small to medium size (500-3000 vertices) and high latency (150 ms+).

CHAPTER 8

LIMITATIONS

Our prototype was developed in Emerald and evaluated on Planetlab. Therefore, the associated results are limited to this environment. Our experiments performed on Planetlab are restricted to the exact nodes that we utilized. We do not attempt to generalize our findings to other environments.

CHAPTER 9

CONCLUSION

Modern computing is plagued by two bottlenecks that contributes to a restriction of performance; the slow access speed of dynamic memory, and that of the slow speed of light. In distributed systems, where nodes are located far from one another and packet exchanges are frequent - the slow speed of light may be a determining factor of performance.

In this thesis we have presented our solution to this problem. The near-far cloud concept is one which aims to alleviate the issue of invocation overheads associated with latency. The incorporation of caching will further improve performance by reducing the frequency of packet-exchange between the nodes. The decision to incorporate caching in graph-structures was made to take advantage of the concept of attached, which motivates the utilization of recursive data types. By the use of attached, both graph transmission and expansion is greatly simplified.

We have implemented a multi-level caching scheme over Planetlab, which is based on the near-far concept. This scheme acts as a cloud-hierarchy consisting of client, proxy and data center. Instead of merely having the client cache data from the proxy, the proxy is also caching data from that of the data center. This way, the proxy keeps expanding to further reduce invocations towards the data center. To demonstrate the effectiveness of this caching scheme, we have compared the performance of a near-far architecture versus that of a standard client-server architecture.

Our findings indicate that the near-far cloud caching scheme contributes to a

significant reduction in amount remote invocations spent in comparison to the client-server caching scheme. Performance increases are dependant on cache size and latency towards the data center.

CHAPTER 10
FUTURE WORK

This chapter will cover what was supposed to be implemented, but were not - given our time constraints. The near-far graph caching scheme could hypothetically be extended to include the concept of smart chained-proxies in conjunction with parallel caching.

10.1 Eviction Policy

Our findings indicate that there exists a point of convergence where caches are small enough to not have a big impact on search overheads, yet big enough to be effective in a caching scheme. A small to medium sized cache with a properly designed eviction policy such as LRU (least recently used) could be employed to take advantage of this. We did not implement this because of the sheer complexity.

For this to work efficiently on smaller caches, one would have to incorporate some type branch prediction system. For instance, one could associate each edge in the graph with a cost that corresponded to the probability of it being traversed. The probability would be based on past data accesses, or access pattern. Machine learning could be utilized in order to establish this pattern, given big enough data sets.

10.2 Smart Chained-proxies

The original plan was to implement a caching scheme consisting of not merely one proxy, but multiple ones. This way, the proxies would form a chain of caches. In terms of the near-far cloud concept, this would increase the total size of the near-cloud. The caches would incorporate an eviction policy such as LRU, and potentially with machine learning. Each subsequent level of cache would contain more data than that of the previous one. This is a clever way to utilize smaller caches to their fullest.

10.3 Parallel caching

The smart chained-proxies scheme may be further extended to include multiple neighbouring caches. A dispatcher will first be consulted upon request for cache-look ups. The dispatcher acts as a monitor object and is responsible for coordinating the neighbouring caches when performing parallel cache look-ups. The whole idea here is to speed up the process of look-ups.

10.4 Summary

This chapter has been a summary of our hopes and dreams for the future of the near-far cloud concept. An introduction to smart chained-proxies and parallel caching have been given.

CHAPTER 11

BIBLIOGRAPHY

[1] Makariel git repository. `https://github.com/Makariel/Near-Far-Cloud-Caching`, May 2017.

[2] Planetlab home page. `https://www.planet-lab.eu/`, March 2017.

[3] Andrew P Black, Norman C Hutchinson, Eric Jul, and Henry M Levy. The development of the emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 11–1. ACM, 2007.

[4] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM.

[5] Michael Byrne. Memory is holding up the moore's law progression of processing power. `https://motherboard.vice.com/en_us/article/memory-is-holding-up-the-moores-law-progression-of-processing-power`, July 2014.

[6] Michele Chinosi and Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[7] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[8] Cryptor. Safenetwork's economics to incentivize different types of memory. https://safenetforum.org/t/safenetworks-economics-to-incentivize-different-types-of-memory/9066, May 2016.

[9] Marios D Dikaiakos, Dimitrios Katsaros, Pankaj Mehra, George Pallis, and Athena Vakali. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet computing*, 13(5), 2009.

[10] Alan Freedman. *Computer desktop encyclopedia.* Osborne/McGraw-Hill, New York, 2001.

[11] Justin O Graver and Ralph E Johnson. A type system for smalltalk. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 136–150. ACM, 1989.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[13] Norman C Hutchinson. Emerald: An object-based language for distributed programming. Technical report, Washington Univ., Seattle (USA), 1987.

[14] Norman C Hutchinson, Rajendra K Raj, Andrew P Black, Henry M Levy, and Eric Jul. *The EMERALD Programming Language: Report.* University of Arizona. Department of Computer Science, 1987.

[15] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.

[16] Ron Maltiel. Forget moore's law: Hot and slow dram is a major roadblock to exascale and beyond. http://semiconductorexpert.blogspot.no/2014/07/bottlenecks-dram-moores-law.html, July 2014.

[17] Christopher Mims. Why cpus aren't getting any faster. https://www.technologyreview.com/s/421186/why-cpus-arent-getting-any-faster/, October 2010.

[18] Razavy Mohsen. Quantum theory of tunneling, 2003.

[19] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Society Newsletter*, 20(3):33–35, 2006.

[20] Paulo Neto. Demystifying cloud computing. In *Proceeding of Doctoral Symposium on Informatics Engineering*, 2011.

[21] R. Penrose. *The Road to Reality: A Complete Guide to the Laws of the Universe*. Vintage Series. Vintage Books, 2007.

[22] John M. Zavada Salah M. Bedair and Nadia El-Masry. Spintronic memories to revolutionize data storage. `http://spectrum.ieee.org/semiconductors/memory/spintronic-memories-to-revolutionize-data-storage`, October 2010.

[23] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[24] Hichael Mareno Søbstad. Hichael's thesis illustrations, 2017.

[25] Dean Takahashi. Forty years of moore's law. `http://www.seattletimes.com/business/forty-years-of-moores-law/`, April 2005.

[26] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java. Mark Allen Weiss.* Pearson Education, 2012.