Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Τίτλος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

MARIOS

Επιβλέπων : test
test

Αθήνα, Ιανουάριος 1111

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Τίτλος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

MARIOS

Επιβλέπων : test
test

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Ιανουαρίου 1111.

.....	.....	.....
test test	test test	test test

Αθήνα, Ιανουάριος 1111

marios

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Λέξεις κλειδιά

Abstract

Key words

Ευχαριστίες

marios,

Αθήνα, 1η Ιανουαρίου 1111

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

List of Figures 13

List of Tables 15

1. Introduction 19
1.1 Thesis motivation 20
1.2 Thesis structure 21

2. Theoretical Background 23
2.1 Distributed storage systems 23
2.2 Archipelago 23
2.3 Tracing Systems 23
2.3.1 Magpie 24
2.3.2 DTrace 24
2.3.3 SystemTap 25
2.3.4 Conclustion 25
2.4 Logging Systems 25

Bibliography 27

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

When back in April 1965 Gordon E. Moore stated the following

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."[9]

had no idea that he had actually started a race among the academia and the industry to overcome or at least abide the this law.

At first, since the technology was premature, the evolution in VLSI technology went hand in hand with the evolution in computer architecture. The more and faster transistors resulted in achievements in instruction level parallelism (ILP). From 1975 to 2005 the endeavour put in computer architecture resulted in technological advances varying from deeper pipelines and faster clock speeds to superscalar architectures. But in around 2005 the ILP wall was hit. Transistors could not be utilized to increase serial performance, logic became too complex and performance attained was very low compared to power consumption. This lead to the creation of multicore systems and entered the programmers to the jungle of parallel software. So far the evolution was almost in accordance with the famous law. However, in around 2009 to 2011, it was the power wall's time to be hit. The famous power equation $P = cV^2 f$ along with the CPU to memory gap (eikona) led to the technological burst of distributed and cloud computing.

In 2009 Amazon.com introduced the Elastic Compute Cloud and since then the term 'cloud' is one of the hottest buzzwords not only among the industry and academia but also among everyday people that take advantage of the 'power of cloud'. Although the term may be vague, the definition of cloud computing, according to NIST (National Institute of Standards and Technology), is the following:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics ,three service models, and four deployment models."[10]

In the previous brief computer chronology, I kept describing bottlenecks and walls to be overcome. However, it not clear how these bottlenecks become obvious and how scientists can be sure that they have reached one's technology's limits before moving on to the next one. The answer to the previous questions has always been given through tracing. Tracing is a process recording information about

a program's execution, while it is being executed. These information may be low level metrics like performance counters or time specific metrics in order to evaluate system's latencies and throughput. Tracing data are mostly useful for developers and can be used for debugging, performance tuning and performance evaluation. From the single-cpu, integrated computer to the hundreds-node cloud infrastructure, trace and performance engineers face challenging problems that vary from platform to platform, but in any case play a vital role the system's design and implementation.

Cloud and distributed computing provided trace engineers with more challenging problems. The system scale is now much greater and program execution is far from deterministic and can take place in any cluster node. So each program execution is not bounded to a specific context. Other problems that needed solving was data and time correlation between the different computing nodes. Also, unlike single chip platforms that can be individually traced and evaluated, cloud infrastructures need to be traced with full-load under production conditions. This set more restrictions concerning the overhead that tracing adds to the application. Finally, tracing is notorious about the amount of data that produces. So distributed and cloud tracing demands the use of distributed data storage systems and processing methods like distributed NOSQL databases and Map-Reduce frameworks.

So to sum up, as described by any design model, the system verification consists a major part of a system's implementation and working process. Verification is achieved through monitoring and tracing. Depending on the system's nature tracing and monitoring process and the tools used may vary. Picking the right tracing tools that will reveal the system's vulnerabilities and faults can be very demanding and the performance engineer for bringing them to light, respecting all the prerequisites set by the system.

1.1 Thesis motivation

The motivation behind this thesis emerged from concerns about the storage performance of the Synnefo [1] cloud software, which powers the ~**okeanos** [2] public cloud service [7]. I will briefly explain what ~**okeanos** and Synnefo are in the following paragraphs.

~**okeanos** is an IaaS (Infrastructure as a Service) that provides Virtual Machines, Virtual Networks and Storage services to the Greek Academic and Research community. It is an open-source service that has been running in production servers since 2011 by GRNET S.A. [3]

Synnefo [8] is a cloud software stack, also created by GRNET S.A., that implements the following services which are used by ~**okeanos** :

- *Compute Service*, which is the service that enables the creation and management of Virtual Machines.
- *Network Service*, which is the service that provides network management, creation and transparent support of various network configurations.
- *Storage Service*, which is the service responsible for provisioning the VM volumes and storing user data.
- *Image Service*, which is the service that handles the customization and the deployment of OS images.

[1] www.synnefo.org/
[2] https://okeanos.grnet.gr/
[3] Greek Research and Technology Network, https://www.grnet.gr/

- *Identity Service*, which is the service that is responsible for user authentication and management, as well as for managing the various quota and projects of the users.

Synnefo provides each virtual machine with at least one virtual volume provisioned by the Volume Service called Archipelago[5] and will be furthered detailed in Chapter . This thesis' purpose is to provide the developer or the system administrations with a cross-layer representation accompanied with the equivalent metrics and time information of an I/O request's route within the infrastructure from the time it is created inside the virtual machine till it is finally served by the storage backend. The design and implementation has to be done respecting the following two prerequisites:

- The tracing information should be gathered and processed in real-time from every node participating in the request serving.
- The tracing infrastructure should add the least possible overhead to the instrumented system, which should continued working properly production-wise

After the end of the tracing infrastructure implementation, the developer should be able to identify the distinct phases and the duration of each that an IO request passes through, measure communication latencies between the different layers and collect all the necessary information (chosen by him) that would help him understand the full context under which this specific request was served. All these information can be used for software faults detection and performance tuning as well as hardware malfunctions and faults like disk or network failures that would be difficult to detect otherwise.

The novelty of this thesis consists in combining live cross-layer, multi-node data aggregation, which is typical for monitoring but not for tracing, with the precision and accuracy of tracing, respecting a hard prerequisite of low overhead. Previous tracing infrastructures offered only partial solutions. Some of them would separate the tracing from the working phase because of the great added overhead, others provided no mechanism for data correlation, while the traditional monitoring systems did not meet our low-level tracing needs.

The proposed system is called *B*lkKin. It is designed respected the aforementioned prerequisites and make use of the latest tracing semantics and infrastructures employed by great tech companies like Google and Twitter.

1.2 Thesis structure

This thesis is structured as follows:

Chapter 2

Theoretical Background

In this chapter we provide the necessary background to familiarize the reader with the main concepts and mechanism used later in the document. For every subsystem employed in BlkKin we briefly describe some counterparts justifying our choice. The approach made is rudimentary, intended to introduce a reader with elementary knowledge on distributed systems.

Specifically, Section 2.1 covers the concepts around distributed storage systems and they difficulties concerning their monitoring. In Section 2.2 we describe Archipelago, Synnefo's Volume Service, and how IO requests initiated within the virtual machine end up being served by a distributed storage system. In Section 2.3 we explain the need for tracing and cite various open-source tracing systems with their advantages and disadvantages. Finally, in Section 2.4 we describe the different needs covered by logging and cite some popular logging systems.

2.1 Distributed storage systems

2.2 Archipelago

2.3 Tracing Systems

Understanding where time has been spent in performing a computation or servicing a request is at the forefront of the performance analyst's mind. Measurements are available from every layer of a computing system, from the lowest level of the hardware up to the top of the distributed application stack. In recent years we have seen the emergence of tools which can be used to directly trace events relevant to performance. This is augmenting the traditional event count and system state instrumentation, and together they can provide a very detailed view of activity in the complex computing systems prevalent today.

Event tracing has the advantage of keeping the performance data tied to the individual requests, allowing deep inspection of a request which is useful when performance problems arise. The technique is also exceptionally well suited to exposing transient latency problems. The downsides are increased overheads (sometimes significantly) in terms of instrumentation costs as well as volumes of information produced. To address this, every effort is taken to reduce the cost of tracing - it is common for tracing to be enabled only conditionally, or even dynamically inserted into the instrumented software and removed when no longer being used.

In early 1994, a technique called dynamic instrumentation or Dyninst API [6] was proposed to provide efficient, scalable and detailed data collection for large-scale parallel applications (Hollingsworth et al., 1994). Being one of the first tracing systems, the infrastructure built for data extraction was limited. The operating systems at hand were not able to provide efficient services for data extraction. They had

to build a data transport component to read the tracing data, using the ptrace function, that was based on a time slice to read data. A time slice handler was called at the end of each time slice, i.e when the program was scheduled out, and the data would be read by the data transport program built on top.

This framework made possible new tools like DynaProf [4] and graphical user interface for data analysis. DynaProf is a dynamic profiling tool that provides a command line interface, similar to gdb, used to interact with the DPCL API and to basically control tracing all over your system.

Kernel tracing brought a new dimension to infrastructure design, having the problem of extracting data out of the kernel memory space to make it available in user-space for analysis. The K42 project [1] used shared buffers between kernel and user space memory, which had obvious security issues. A provided daemon waked up periodically and emptied out the buffers where all client trace control had to go through. This project was a research prototype aimed at improving tracing performance. Usability and security was simply sacrificed for the proof of concept. For example, a traced application could write to these shared buffers and read or corrupt the tracing data for another application, belonging to another user.

In the next sections, recent tracers and how they built their tracing infrastructure will be examined.

2.3.1 Magpie

One of the earliest and most comprehensive event tracing frameworks is Magpie [2]. This project builds on the Event Tracing for Windows infrastructure which underlies all event tracing on the Microsoft Windows platform. Magpie is aimed primarily at workload modelling and focuses on tracking the paths taken by application level requests right through a system. This is implemented through an instrumentation framework with accurate and coordinated timestamp generation between user and kernel space, and with the ability to associate resource utilisation information with individual events.

The Magpie literature demonstrates not only the ability to construct high-level models of a distributed system resource utilisation driven via Magpie event tracking, but also provides case studies of low-level performance analysis, such as diagnosing anomalies in individual device driver performance. Magpie utilises a novel concept in behavioural clustering, where requests with similar behaviour (in terms of temporal alignment and resource consumption) are grouped. This clustering underlies the workload modelling capability, with each cluster containing a group of requests, a measure of “cluster diameter”, and one selected “representative request” or “centroid”. The calculation of cluster diameter indicates deep event knowledge and inspection capabilities, and although not expanded on it implies detailed knowledge of individual types of events and their parameters. This indicates a need for significant user intervention to extend the system beyond standard operating system level events.

As an aside, it is worth noting here that, for the first time, we see in Magpie the use of a binary tree graph to represent the flow of control between events and sub-events across distinct client/server processes and/or hosts.

2.3.2 DTrace

Then, Sun Microsystemsa released, in 2005, DTrace [3] which offers the ability to dynamically instrument both user-level and kernel-level software. As part of a mass effort by Sun, a lot of tracepoints were added to the Solaris 10 kernel and user space applications. Projects like FreeBSD and NetBSD also ported dtrace to their platform, as later did Mac OS X. The goal was to help developers find serious performance problems. The intent was to deploy it across all Solaris servers and to use it in production. If we look at the DTrace architecture, it uses multiple data providers, which are basically probes used to gather tracing data and write it to memory buffers. The framework provides a user space

library (libdtrace) which interacts with the tracer through ioctl system calls. Through those calls, the DTrace kernel framework returns specific crafted data for immediate analysis by the dtrace command line tool. Thus, every interaction with the DTrace tracer is made through the kernel, even user space tracing. On a security aspect, groups were made available for different level of user privileges. You have to be in the dtrace proc group to trace your own applications and in the dtrace kernel group to trace the kernel. A third group, dtrace user, permits only system call tracing and profiling of the user own processes. This work was an important step forward in managing tracing in current operating systems in production environment. The choice of going through the kernel, even for user space tracing, is a performance trade-off between security and usability.

2.3.3 SystemTap

In early 2005, Red Hat released SystemTap [11] which also offers dynamic instrumentation of the Linux kernel and user applications. In order to trace, the user needs to write scripts which are loaded in a tapset library. SystemTap then translates these in C code to create a kernel module. Once loaded, the module provides tracing data to user space for analysis. Two system groups namely stapdev and stapusr are available to separate possible trac- ing actions. The stapdev group can do any action over Systemtap facilities, which makes it the administrative group for all tracing control (Don Domingo, 2010) and module creation. The second group, stapusr, can only load already compiled modules located in specific protected directories which only contain certified modules. The project also provides a compile-server which listens for secure TCP/IP connections using SSL and handles module compilation requests from any certified client. This acts as a SystemTap central module registry to authenticate and validate kernel modules before loading them. This has a very limited security scheme for two reasons. First, privileged rights are still needed for specific task like running the compilation server and loading the modules, since the tool provided by Systemtap is set with the setuid bit. Secondly, for user space tracing, only users in SystemTap's group are able to trace their own application, which implies that a privileged user has to add individual users to at least the stapusr group at some point in time, creating important user management overhead. It is worth noting that the compilation server acts mostly as a security barrier for kernel module control. However, like DTrace, the problem remains that it still relies on the kernel for all tracing actions. Therefore, there is still a bottleneck on performance if we consider that a production system could have hundreds of instrumented applications tracing simultaneously. This back and forth in the kernel, for tracing control and data retrieval, cannot possibly scale well.

2.3.4 Conclustion

It is obvious from the previous analysis that the systems mentioned do not fit in our demands concerning the added overhead to the instrumented application since the solution pass through the kernel space. This makes them unsuitable for live tracing. The solution for for the BlkKin tracing backend was given from the Linux Trace Toolkit - next generation (LTTng) because it provides separate mechanisms for kernel and user space tracing. LTTng is furthered examined in Chapter 1.1.

2.4 Logging Systems

Bibliography

[1] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42's Performance Monitoring and Tracing. IBM Research.

[2] Paul Barham, Rebecca Isaacs, Richardand Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In Proceedings of the 9th conference on Hot Topics in Operating Systems, 2003.

[3] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instru- mentation of production systems. USENIX.

[4] Luize Derose, Jefrey K. Hollingsworth, and Ted Hoover Jr. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In Parallel and Distributed Processing Symposium., Proceedings 15th International., 2001.

[5] Filippos Giannakos, Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos large-scale cloud service using cepg. ;login, 39(3):6–9, June 2014.

[6] Jefrey K. Hollingsworth, Barton P. MILLER, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In Proceeding of the Scalable High Performance Computing Conference, 1994.

[7] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos: Building a cloud, cluster by cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[8] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. Synnefo: A complete cloud stack over ganeti. ;login, 38(5):6–10, October 2013.

[9] Gordon E. Moore. Cramming more components onto integrated circuits. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, Readings in computer architecture, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[10] National Institute of Standards and Technology. The NIST definition of cloud computing. Special Publication 800-145, September 2011.

[11] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brand Chen. Locating System Problems Using Dynamic Instrumentation. In Proceedings of the Ottawa Linux Symposium, 2005.