Karlsruhe Institute of Technology

Institut für Technische Informatik
Lehrstuhl für Rechnerarchitektur
Prof. Dr. rer. nat. Wolfgang Karl

Integration of Safety-Critical Real-Time Applications on Platform Virtualization Software

Diplomarbeit
von

Aurelien Lourot

an der Fakultät für Informatik

Tag der Anmeldung: 2. November 2009
Tag der Abmeldung: 30. April 2010

Aufgabensteller:
Prof. Dr. rer. nat. Wolfgang Karl

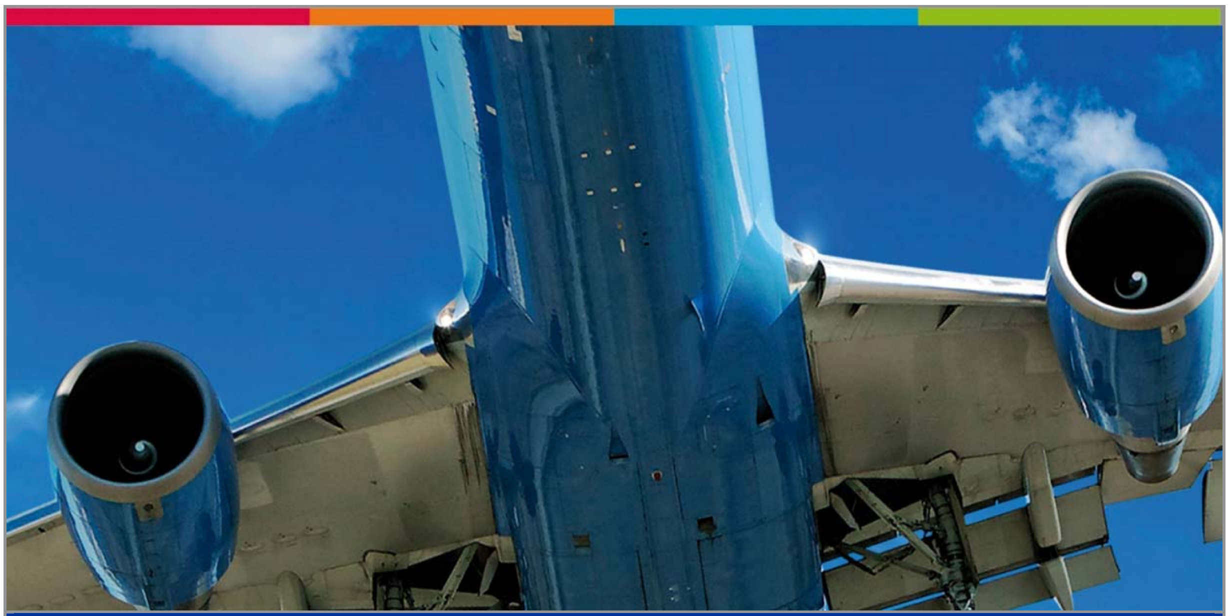
Betreuer:
Dipl.-Inform. David Kramer
Dipl.-Inform. Frank Kulasik

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 30.04.2010

____

Aurelien Lourot



Diploma Thesis

Integration of Safety-Critical Real-Time Applications on Platform Virtualization Software

Presented by Aurelien Lourot

V1.0/30.04.2010

www.kit.edu

if.insa-lyon.fr

www.comsoft.aero

Revision History

Version	Date	Description	Resp.
V1.0	30.04.2010	Initial Version	A. Lourot

Acknowledgements

First of all I would like to thank Prof. Dr. rer. nat. Wolfgang Karl (KIT), David Kramer (KIT), Frank Kulasik (Comsoft) and Mathieu Maranzana (INSA Lyon) for supervising this diploma thesis.

Furthermore, I would like to show my gratitude to several colleagues and friends in the context of this project: Andrew Dale, Johannes Fischer, Andreas Holzmann, Dr. Bernhard Limbach, Dr. Klaus Lindemann, Frank Muzzulini, Jonas Weismüller and Marc Winkelmann for their help concerning the Comsoft products; Manfred Kissel, Peter Risse and Thomas Wirth for their help concerning hardware; David Barton, Yasmin Jakober and Isabelle Kurpat for their support concerning the English and for proofreading this thesis; Dieter Drabold for his training course on electrostatic discharge.

Last but not least I would like to thank all my colleagues who are not explicitly mentioned, for their kindness and for making the company Comsoft a good place to work.

Abstract (German)

Die Firma Comsoft GmbH beliefert ihre Kunden aus dem Bereich Air Traffic Management mit Komplettsystemen. Ein System ist in diesem Fall ein Set von mehreren weichen Echtzeitsanwendungen, die auf einem Set von Hardware Plattformen laufen. Sie brauchte eine Lösung für Serverkonsolidierung mit der Möglichkeit, die Ressourcen (CPU, Festplatte, usw.) gleichmäßig zwischen den Anwendungen zu verteilen, und mit einer garantierten Beschränkung der von einer Anwendung zur Erreichung einer Ressource gebrauchten Zeit. Comsoft bestellte eine auf Servervirtualisierung basierende Studie.

Wir haben die Anforderungen an dieser Lösung beschrieben und einen Entwurf vorgeschlagen. Wir haben eine Reihe von Indikatoren und Tests definiert und haben sie verwendet, um die Vorhersagbarkeit, die Leistung und die Isolierungskapazität existierender Virtualisierungsansätze genau zu bewerten. Wir haben einige Produkte ausgewählt, die die Anforderungen von Comsoft erfüllen könnten, und haben unsere neulich definierte Methodik verwendet, um ihre Fähigkeit oder Unfähigkeit bei der Unterstützung und Isolierung der Echtzeitsanwendungen Comsofts zu beweisen und damit die Auflösung des vorgelegten Problems auch zu beweisen. Die Tauglichkeit von VMware ESXi wurde bewiesen.

Wir haben eine auf ESXi basierende Lösung umgesetzt und geprüft, um die Erfüllung der übrigen Anforderungen zu bestätigen.

Gewerblich gesehen ist die Besonderheit dieser Studie die Sicherheitskritizität des Projekts. Wissenschaftlich gesehen ist diese Arbeit unseres Wissens die erste Definition einer Methodik zur Evaluierung existierender Virtualisierungsansätze, die sich so eingehend mit Vorhersagbarkeit und Echtzeit bei allen virtuellen Ressourcen beschäftigt.

Abstract (English)

The Comsoft GmbH Company supplies their customers with full systems for Air Traffic Management. Such a system is actually a collection of several soft real-time applications running on a set of hardware platforms. The company needed a solution for server consolidation with the possibility to equally distribute resources (CPU, hard-disk, etc.) among their applications and the guarantee that the time needed for an application to get a resource is limited. They ordered a study based on server virtualization.

We described the requirements for such a solution and proposed a design. We defined a series of indicators and tests, and used them to precisely evaluate predictability, performance and isolation of existing virtualization products. We selected virtualization products that may fulfil Comsoft's requirements and used our newly defined methodology to prove their ability or inability to support and isolate Comsoft's soft real-time applications, and therefore to solve the submitted problem. The suitability of VMware ESXi has been proved.

We implemented a solution based on ESXi and tested it to confirm that the remaining requirements are met.

From an industrial point of view, the particularity of this study is its safety-critical nature. From a scientific point of view, this is, to our knowledge, the first definition of a methodology for evaluating existing virtualization products focusing so precisely on predictability and real-time for all virtualized resources.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The Comsoft GmbH Company [A1] delivers *solutions* for Air Traffic Management[1] (ATM) to its customers. Such a solution is actually a collection of several full products (also called *systems*), each of them being a collection of software processes (also called *subsystems*) running on a set of hardware *platforms* (computers).

Note: One instance of a subsystem can only run on <u>one</u> platform.

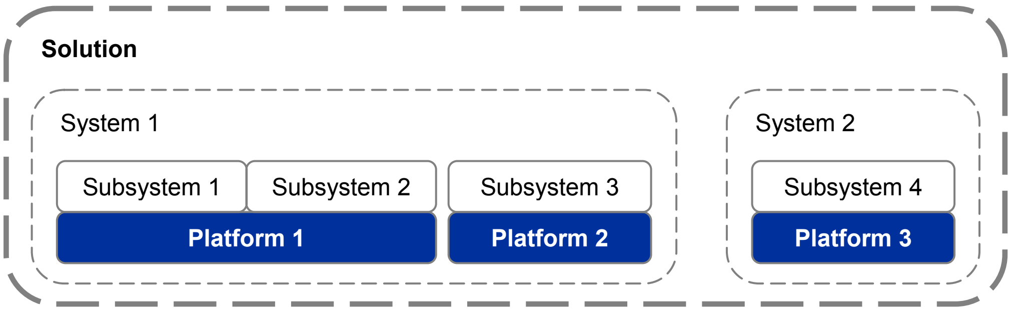


Figure 1-1: Example of a solution

For obvious commercial reasons, Comsoft would like to be able to reduce the amount of platforms. In this process, following elements have to be taken into account:

- The subsystems are executed on different and dedicated hardware platforms. This isolation reduces drastically the amount of (unexpected) interaction between them, thus helping to guarantee that each of them (and therefore the whole solution) fulfils the performance requirements (see chapter 2.3 Requirements). The suppression of this isolation often leads to a violation of these requirements, since the different subsystems compete for the access to the same hardware components[2].
- In the context of ATM, *safety* (protection against harm and damage) is an important notion. The performance requirements are actually just a subset of the safety requirements. In this context, a few tests are not sufficient to validate a solution and it has to be demonstrated that the requirements are met.

[1] Air Traffic Control (ATC), Air Traffic Flow Management (ATFM), Air Space Management (ASM), Air Safety…

[2] Not only the processor, but also the memory and all I/O components (hard disk drive, network peripherals, …).

- One important safety requirement is to avoid a *single point of failure*[1] (SPOF). In consequence each component of the system exists at least twice, leading to this type of solution:

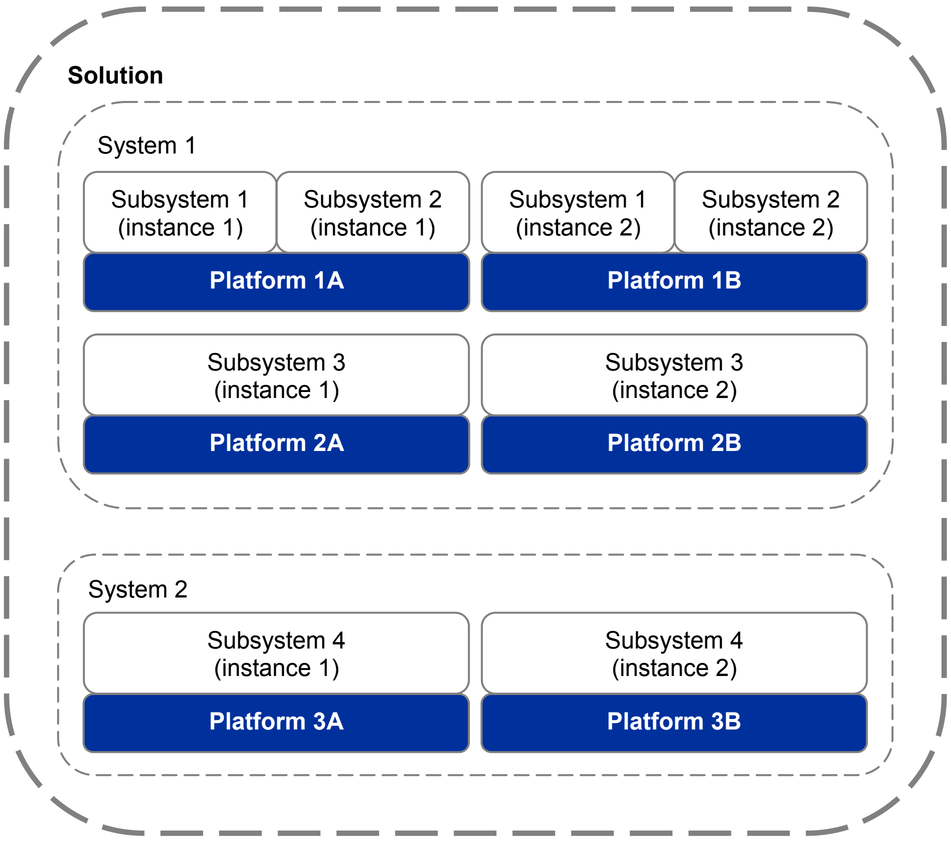


Figure 1-2: Previous solution without SPOF

Note: The different instances of the same subsystem are aware of each other and work together by means of clustering mechanisms, thus increasing the *availability*[2] of the solution.

For now, Comsoft doesn't sell any solution where several substantial systems share the same platforms for the following reasons:

- The proof that their software applications fulfil the safety requirements is so far made only in cases where each application runs on a dedicated platform and is considered as a full independent product.
- Some tests were performed internally, where several applications were run on the same platform (without virtualization layer), but were not always satisfactory. (Some examples are given in chapter 2.1.1 Comsoft products.)

[1] Component of a system which, if it fails, prevents the whole system from working. [N1]
[2] The availability of a system during a given time interval is the probability that the system is able to deliver the service(s) it is designed for. For instance an availability of 99% means that the system will probably not work **as expected** (unavailability during a planned maintenance process doesn't count) 3.65 days a year. [H1]

However making several systems share the same platforms is the goal of this diploma thesis. A solution should then have the following structure:

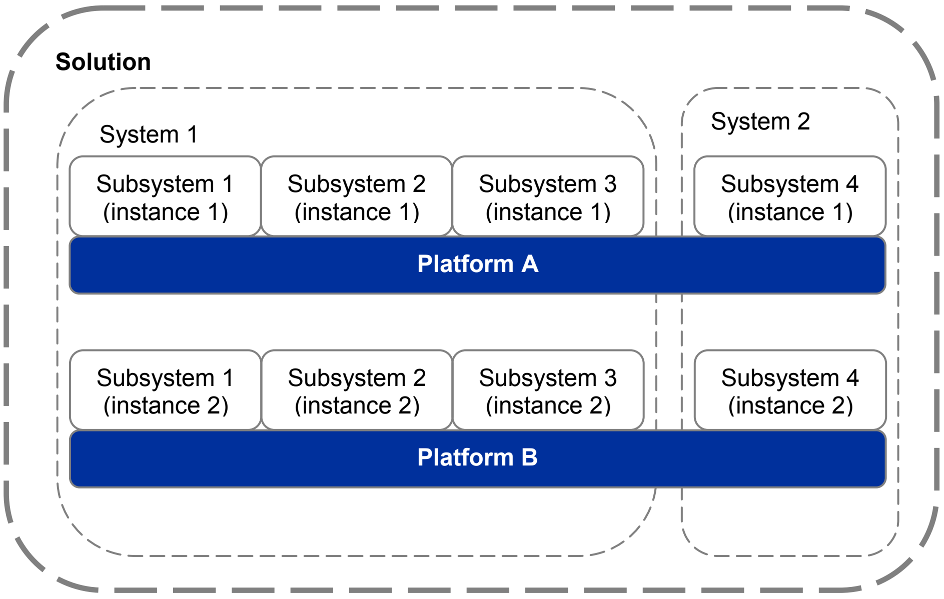


Figure 1-3: Previous solution with the least possible platforms (still without SPOF)

Some precisions concerning the current Comsoft's current solutions have to be made at this point:

- Each hardware platform is an HP Industrial Server of the HP ProLiant DL series or a successor. Since these servers have proven their worth after years of utilization and HP guaranties a support all over the world, they shall still be deployed.
- The operating system is a Red Hat or Fedora Linux distribution. Comsoft shall keep using them, because a safety assessment was already performed for those distributions.
- The systems are AIDA-NG, CADAS-ATS and CADAS-IMS (see chapter 2.1.1 Comsoft products), each of their software parts consisting of several million lines of code, as well as some smaller complementary products such as an X.500 directory server for instance.
- Those systems are restricted by time constraints and can be, as we will see further on in this document (see chapter 2.1.1 Comsoft products), considered as *soft real-time systems*.

1.2 Objective

As mentioned, trying to deploy several subsystems on the same hardware platform often leads to a failure to fulfil the safety requirements (see chapter 2.3 Requirements). For instance CADAS-ATS performs a database replication, grabbing the CPU and RAID-I/O in such a manner that AIDA-NG gets temporarily blocked and the message handling's latency exceeds a threshold.

Setting up an adapted *process scheduling policy* is therefore not sufficient. Memory management and I/O-access must be studied, and mutual blocking of applications must be temporally limited.

It seems obvious, that the access from applications to common resources such as CPU, RAM and I/O devices could be easily controlled by virtualization techniques (see chapter 2.1.2 Virtualization). Instead of deploying subsystems on *real platforms*, it would be possible deploy them on *virtual platforms* (representing a fraction of a real platform). No modification on the subsystems (i.e. no software modification) would be needed:

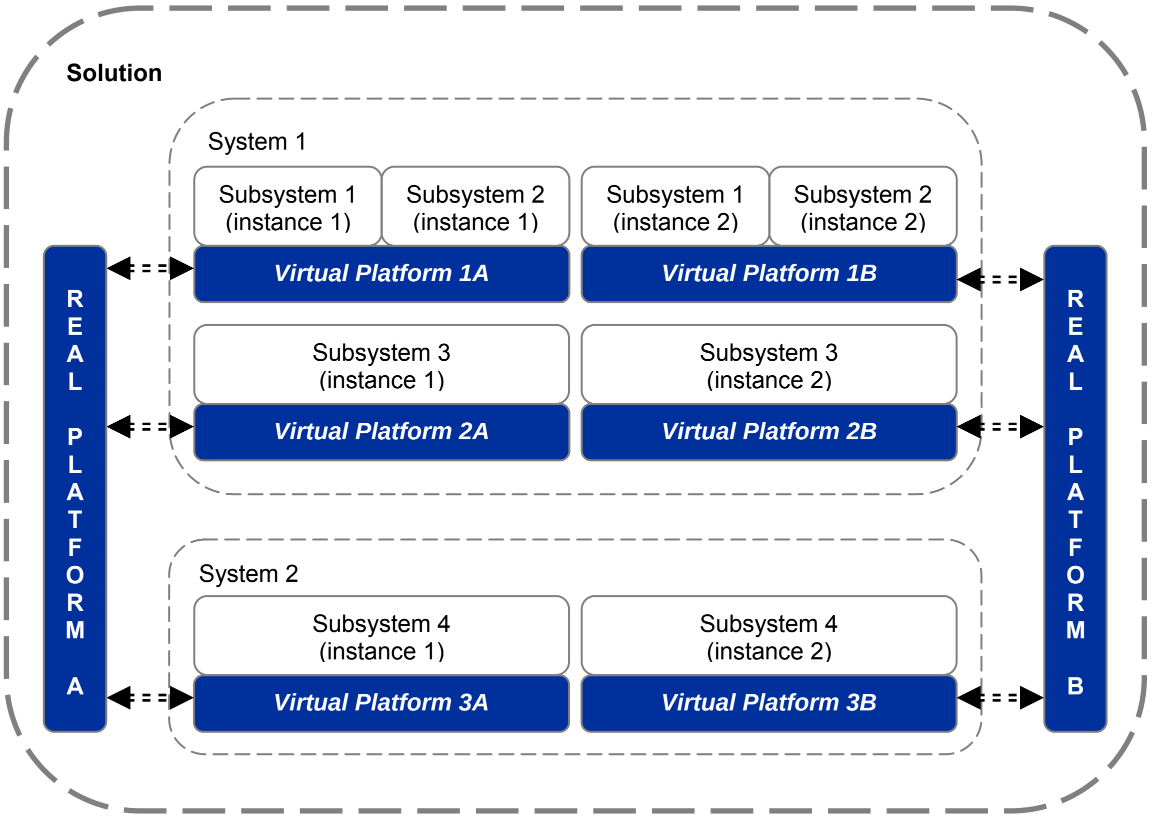


Figure 1-4: Previous solution with virtualization (still without SPOF)

Note: There is still no *single point of failure* in the figure 1-4, since there is no subsystem whose instances are on the same **real** platform.

1.3 Tasks

A virtualization software fulfilling the safety and commercial requirements has to be deployed. As a result, the tasks to be performed in this diploma thesis are as follows:

- The safety and commercial requirements have to be clearly specified and justified.
- These requirements have to be completed by other requirements resulting from existing applications and components' characteristics (compatibility of virtualization software with hardware and operating system, with clustering mechanisms, etc.).
- Existing virtualization techniques and corresponding implementations shall be evaluated according to the requirements and their validity shall be tested.
- The target architecture has to be designed for the found valid techniques and implementations.

- A system corresponding to this architecture shall be implemented. This system should be suitable for an operational deployment. This means for example that no hardware other than the one usually needed can be used.
- Test plans derived from the previous requirements must be specified and executed. The traceability from requirement to test plan must be clear.
- In case a technical argumentation in addition to test plans is necessary or meaningful to prove the fulfilment of a requirement, this argumentation shall be done.
- The fulfilment (or non-fulfilment) of the requirements shall also be examined with an overall view.
- If residual problems are found in the overall view, the tasks to solve those problems shall be specified.

1.4 Structure of the Thesis

Chapter 2 first gives technical explanations on Comsoft products, virtualization and real-time systems, needed to understand the next parts of this document. It then presents the research work related to our study and the requirements for the solution we propose.

We expose in chapter 3 the design of our solution and evaluate virtualization products in order to implement it. This evaluation is the main part of our study.

We explain our implementation in chapter 4. Chapter 5 describes the verification and the validation of this implementation.

Finally, chapter 6 sums up the results achieved while working on this thesis and points out future work and perspectives.

2 Background

2.1 Technical Background

This chapter gives technical explanations needed to understand the rest of this document.

2.1.1 Comsoft products

This chapter describes the Comsoft products (also called systems, see chapter 1.1 Motivation) concerned by this project.

AIDA-NG

AIDA-NG stands for *Aeronautical Integrated Data Exchange Agent – Next Generation*. [A5] This product is a router for aeronautical messages (flight plans, weather reports, etc.) in ground data networks. It supports most communication protocols of all layers used all over the world for aeronautical telecommunication and is able to convert messages from one standard to another, thus acting as a gateway too.

The two main aeronautical interconnection networks (covering the Earth) are:

- AFTN: *Aeronautical Fixed Telecommunication Network*. It involves *AFTN messages* and *AFTN addresses*. It is an "old" standard (1950s).
- AMHS: *ATS (Air Traffic Services) Message Handling System*, with so called *AMHS messages* and *AMHS addresses*. It is a newer standard (1990s) aiming to replace AFTN.

Their lines are very heterogeneous. They are based on a lot of different protocol stacks (X.400 over IP, X.400 over X.25, TCP/IP, UDP/IP, etc.).

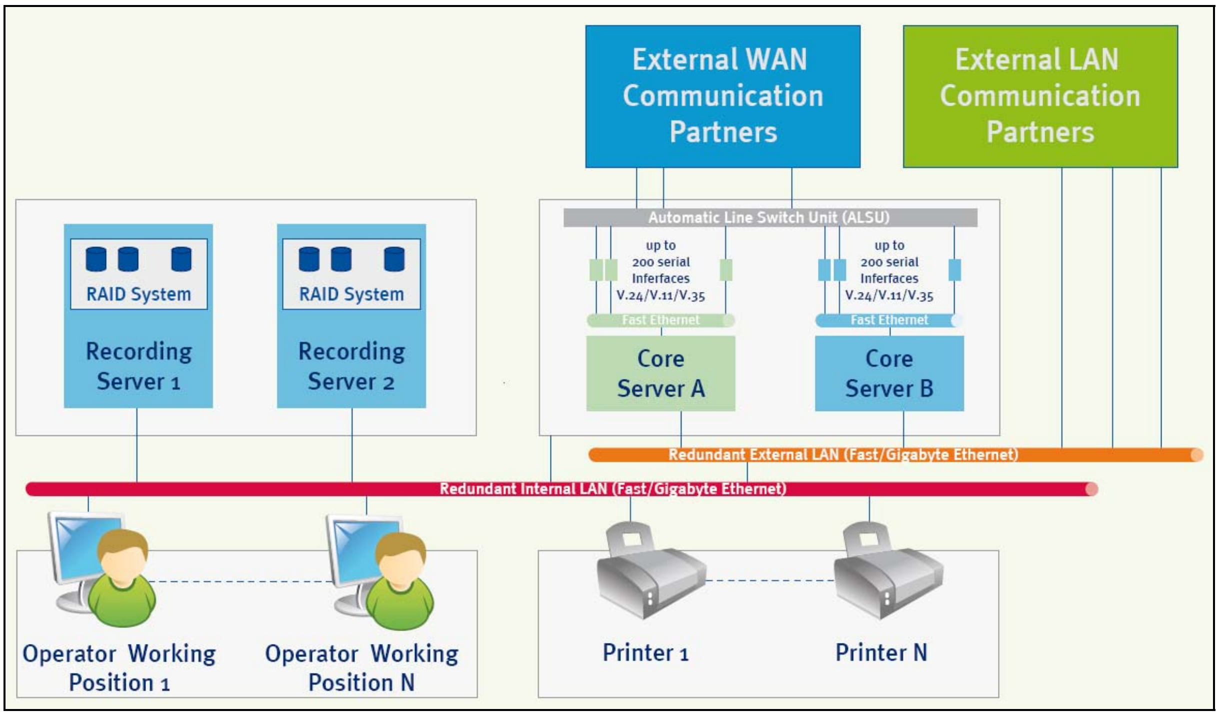

External WAN Communication Partners
External LAN Communication Partners
Automatic Line Switch Unit (ALSU)
up to 200 serial Interfaces V.24/V.11/V.35
up to 200 serial Interfaces V.24/V.11/V.35
Fast Ethernet
Fast Ethernet
Core Server A
Core Server B
RAID System
Recording Server 1
RAID System
Recording Server 2
Redundant External LAN (Fast/Gigabyte Ethernet)
Redundant Internal LAN (Fast/Gigabyte Ethernet)
Operator Working Position 1
Operator Working Position N
Printer 1
Printer N

Figure 2-1: Architecture of AIDA-NG

This system is composed of several sub-systems:

- *Core Sub-Systems* (CSS, represented as "Core Server" on figure 2-1): this sub-system routes the messages coming from one communication partner to another, eventually performing conversions, based on routing rules. There are two instances of CSS (CSS A and CSS B), running on different hardware platforms to ensure redundancy.
- *Recording Sub-System* (RSS, represented as "Recording Server" on figure 2-1): this sub-system records in a database all messages going through the system and all events happening in the system. Every airport has the legal obligation to keep a copy of all messages during a time between 30 and 90 days. There are two instances of RSS (RSS 1 and RSS 2), running on different hardware platforms to ensure redundancy. Usually, CSS A and RSS 1 are running on the same hardware platform, CSS B and RSS 2 on another one.
- *Operating Sub-System* (OSS, represented as "Operator Working Position" on figure 2-1): this sub-system is a graphical interface to configure and monitor the whole system. Up to 100 instances of OSS can be deployed. They are usually deployed on workstations (classical PCs). This is why the OSS is not part of this project, which concerns the virtualization of the servers.

Like other Comsoft systems, AIDA-NG relies on two local networks:

- *Internal LAN* (ILAN): interconnects all AIDA-NG components.
- *External LAN* (ELAN): connects AIDA-NG to the other systems of the whole solution.

All components of both networks are doubled, to ensure redundancy. For more information on this system, please refer to the following documents:

- *AIDA-NG Product Information*; [A5]
- *AIDA-NG Interface Control Document*; [A6]

- *AIDA-NG Routing*; [A7]
- *AIDA-NG System Architecture*. [A8]

CADAS-ATS

CADAS-ATS stands for *Comsoft's Aeronautical Data Access System – Air Traffic Services*. [A9] This system allows the creation of AFTN and AMHS mailboxes, so that operators can send and receive AFTN and AMHS messages. This system is connected to the AIDA-NG system (or to another AFTN/AMHS gateway) over the External LAN (see previous chapter). The graphical interface of this system has a lot of standard message templates so that operators can easily write flight plans, meteorological messages, etc.

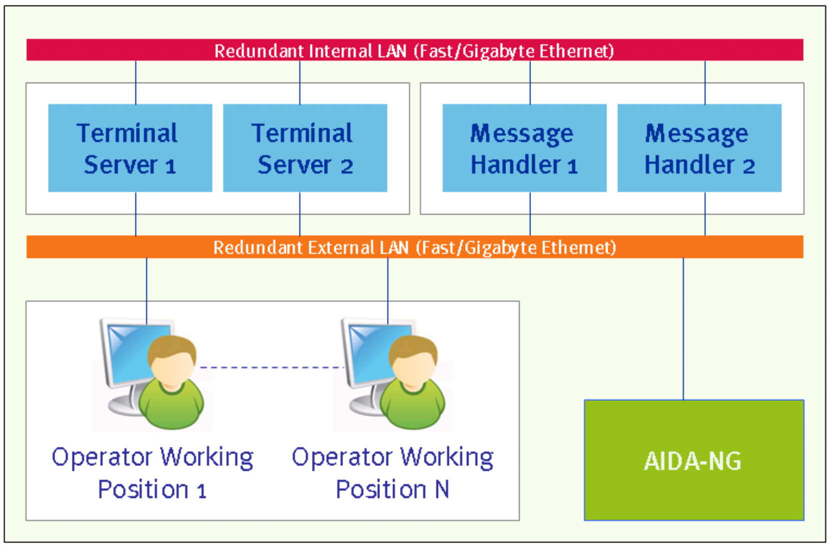

Redundant Internal LAN (Fast/Gigabyte Ethernet)
Terminal Server 1
Terminal Server 2
Message Handler 1
Message Handler 2
Redundant External LAN (Fast/Gigabyte Ethernet)
Operator Working Position 1
Operator Working Position N
AIDA-NG

Figure 2-2: Architecture of CADAS-ATS

The system is composed of several sub-systems:

- *Message Handler*: this sub-system routes the messages to the different mailboxes or to AIDA-NG and stores in a database all messages and all events happening on the system (and keeps them for a time between 30 and 90 days). There are two instances of Message Handler (Message Handler 1 and Message Handler 2), running on different hardware platforms to ensure redundancy.
- *Terminal Server*: this sub-system makes the java applications (*terminals*) available for the Operator Working Positions. There are two instances of Terminal Server (Terminal Server 1 and Terminal Server 2), running on different hardware platforms to ensure redundancy. Usually, Message Handler 1 and Terminal Server 1 are running on the same hardware platform, as well as Message Handler 2 and Terminal Server 2.

Like other Comsoft systems, CADAS-ATS relies on two local networks:

- *Internal LAN* (ILAN): interconnects all CADAS-ATS servers and allows them to synchronise at any time with each other.
- *External LAN* (ELAN): connects CADAS-ATS to the other systems of the whole solution (e.g. AIDA-NG) and to the workstations (Operator Working Positions).

For more information on this system, please refer to the following documents:

- *CADAS-ATS Product Information*; [A9]
- *CADAS-ATS: Administrator's Guide*. [A10]

CADAS-IMS

CADAS-IMS stands for *Comsoft's Aeronautical Data Access System – Information Management & Services*. [A11] There are several types of aeronautical messages. Three of them concerned by CADAS-IMS are *FPL* (Flight Plan), *NOTAM* (Notice to Airmen) and *OPMET* (Operational Meteorological). NOTAMs are messages sent by government agencies to *NOTAM offices* (usually located in airports) to inform pilots of any hazard like for example:

- runway closure due to maintenance;
- military exercises;
- volcanic ash.

A NOTAM can indicate a new hazard, can replace/update a previous one or indicate that a previous NOTAM has to be removed. Without any information system, as it's still the case in some countries, the NOTAM office has a set of cabinets and folders containing one sheet per NOTAM. NOTAM office employees have to regularly add, replace and remove sheets from their "database".

Before a plane takes off, its pilot has to go to the NOTAM office to get a *briefing*. A briefing is the list of all NOTAMs (and OPMETs) concerning its flight (inside its *air corridor*). Its air corridor is already known based on the *FPL* (Flight Plan). A folder is given to the pilot, who can then get onto the plane.

The role of CADAS-IMS is to replace the cabinets, the folders and the sheets of paper with a database, automatically maintain it and automate the creation of *briefings* by cross-referencing the FPLs, the NOTAMs and the OPMETs. This system is connected to the AIDA-NG system (or to another AFTN/AMHS gateway) over the External LAN.

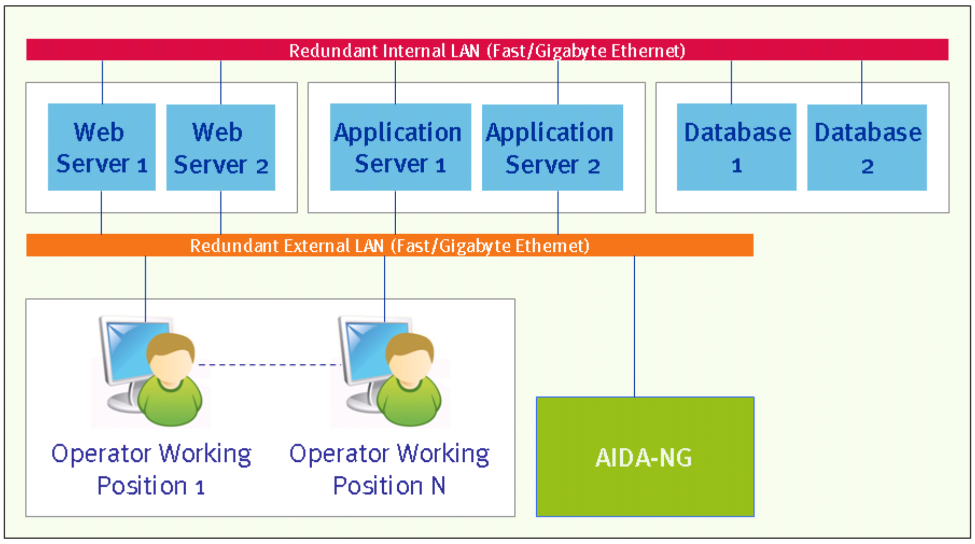

Redundant Internal LAN (Fast/Gigabyte Ethernet)
Web Server 1
Web Server 2
Application Server 1
Application Server 2
Database 1
Database 2
Redundant External LAN (Fast/Gigabyte Ethernet)
Operator Working Position 1
Operator Working Position N
AIDA-NG

Figure 2-3: Architecture of CADAS-IMS

The system is composed of several sub-systems:

- *Database*: this sub-system is the database for all messages and events happening on the system, and for the *static data* (position and boundaries of all countries, position of every airport, standard air corridors, etc.) There are two instances of Database (Database 1 and Database 2), running on different hardware platforms to ensure redundancy.
- *Application Server*: this sub-system contains all applications of the system and executes them. There are two instances of Application Server (Application Server 1 and Application Server 2), running on different hardware platforms to ensure redundancy.
- *Web Server*: this sub-system makes the graphical interface available to the workstations (Operator Working Positions), which only need a web browser to display it. Usually, Database 1, Application Server 1 and Web Server 1 are running on the same hardware platform, Database 2, Application Server 2 and Web Server 2 on another one.

Like other Comsoft systems, CADAS-IMS relies on two local networks:

- *Internal LAN* (ILAN): interconnects all CADAS-IMS servers and allows them to synchronise at any time with each other.
- *External LAN* (ELAN): connects CADAS-IMS to the other systems of the whole solution (e.g. AIDA-NG) and to the workstations (Operator Working Positions).

For more information on this system, please refer to the following documents:

- *CADAS-IMS Product Information*; [A11]
- *CADAS-IMS: Technical Specification*. [A12]

CCMS

CCMS stands for *Comsoft Configuration Management Suite*. [A13] It isn't a product but it's part of all air traffic control solutions delivered by Comsoft. CCMS is a set of applications installed on *all* platforms (servers and workstations). CCMS:

- provides a graphical interface, where administrators can declare and configure all *platforms*, *systems* and *sub-systems* (see chapter 1.1 Motivation);
- spreads the software and the defined configuration to all platforms of the solution;
- tweaks the look and feel of the underlying operating system (Fedora) to work smoothly with Comsoft applications and present a common user interface;
- provides scripts for easier access to Linux functions.

Thus CCMS is installed on all platforms and simplifies their configuration. Administrators don't have to install and configure each platform one by one.

Other products

The following products are for now not concerned by this project but will probably benefit from its results in a near future:

- *CNMS (Comsoft Network Management System)* is a system to supervise and monitor the health of all Comsoft systems at the same time, i.e. the health of all components of the whole solution. [A15]
- *EFG (E-Mail/Fax Gateway)* is a system that receives e-mails, converts them, and sends them as fax messages to the indicated destination. [A16]
- *ATN-Router* (*Aeronautical Telecommunication Network Router)* is a CLNP (*Connectionless Network Protocol)* router. [A17]
- *View500 (formerly ViewDS)* is an X.500 Directory Server from the company eB2Bcom, delivered by Comsoft. [N2]
- *CADAS-AIM$_{DB}$ (Database for Aeronautical Information Management)* is a database containing all data necessary for the safety, regularity and efficiency of international air navigation. It stores entities, attributes and relationships in order to describe aeronautical features such as airports, runways, obstacles, routes, terminal procedures, airspace structures, services and related aeronautical data. [A2] [A18]

Without virtualization

As mentioned in chapter 1.1 Motivation, this project isn't the first attempt from Comsoft to let several systems share the same hardware platforms. No problem has been observed concerning the CPU utilization, the main memory or the network, but rather concerning the hard disk drives.

Every subsystem of AIDA-NG is composed of several processes called *managers*, each of them performing a special task. All managers must send regularly (typically every third second) an *alive message* to a special manager named *system manager*. There is one *system manager* per *subsystem*. After a certain amount of missing alive messages (typically six) from a certain manager, its *system manager* considers this manager to be malfunctioning, stops all *managers* under its authority and declares the subsystem to be *in maintenance*. An administrator must intervene.

The *Recording Subsystem* (RSS) of AIDA-NG is responsible for the database of AIDA-NG and therefore often accesses the hard-disk drive. The Linux hard disk scheduler, trying to optimize the movements of the hard disk (see chapter 3.5.1.3 Hard Disk Drive), doesn't deliver any guarantee concerning the latency of a hard disk access by a process. If one process is using the hard disk drive, another one also trying to access it can get blocked during several seconds. But if one manager of the RSS gets blocked more than about 15 seconds, or needs more than 15 seconds to perform an elementary sequence of hard-disk operations, the whole subsystem gets stopped, which is not acceptable.

It's easy to make this problem happen, following tests can be made:

- copying a "big" file (with the command `dd` or `cp`) during seconds while the RSS is storing a lot of messages will make it crash;
- uncompressing a "big" tar-archive during seconds will lead to the same result.

Comsoft engineers noticed easily that this problem is not related to the maximum bandwidth of the hard disk drive but just to the latency. They were ready to sacrifice the bandwidth in order to guarantee a correct latency (a latency of even one second would have been acceptable!) and fairness. They invested a lot of effort in trying different disk schedulers[1] and configuring them, but did not manage to reduce the latency.

In the field, the Comsoft project for *Vietnam Air Navigation Services Corporation* (VANSCORP) suffered the consequences of this problem. Comsoft delivered a system where AIDA-NG, CADAS-ATS and CADAS-IMS share the same hardware platforms, without virtualization layer. If one Message Handler of CADAS-ATS is switched off during a long period and then switched on again, it has to synchronise its database with the other Message Handler. This process is called a *database replication*. If there is a big difference between the two databases, the replication can take a long time and make the RSS, running on the same hardware platform, wait for the hard disk and crash. It actually happened several times, forcing Comsoft to redesign the solution and deliver more servers.

All Comsoft products are based on such timers, guaranteeing that the solution works correctly in the allotted time, with some tolerances. Thus Comsoft systems can be considered as *soft real-time systems* (see chapter 2.1.3.2 Soft Real Time). The time granularity/precision of Comsoft systems usually varies from one to a few seconds.[2]

Another problem appeared on the same project (VANSCORP). A software product from another manufacturer had to be integrated to the project, but this manufacturer provides an assistance only if the software runs on Fedora Core 4. Because Comsoft products were not based on Fedora Core 4 at that time, an additional server had to be delivered especially for this external software product. This could have been solved by virtualization (see chapter 2.1.2.1 Reasons). However this thesis doesn't focus on this purpose but rather on *server consolidation* and *real-time*.

2.1.2 Virtualization

Virtualization is a set of software and hardware techniques allowing a *real computer* (or real machine) to act like one or more computers (having the same architecture or not). These simulated computers are usually called *(guest) virtual machines*.

This term has also been extended to higher levels: for example an operating system simulating another one is a type of virtualization too (*operating system API emulation*, see chapter 2.1.2.2 Virtualization Techniques).

2.1.2.1 Reasons

This chapter describes the main reasons for using virtualization. More details can be found in *Das Virtualisierungs-Buch* [V1] or *The Advantages of Using Virtualization Technology in the Enterprise* [V9].

Server consolidation

Some applications sometimes need to be separated on different servers, because for example they need different operating systems, or several instances of the same application can't coexist in the same environment. Several computers are then needed. It often leads to a waste of resources, since the whole capacity of a computer is rarely used.

[1] Linux proposes four different schedulers: *CFQ*, *Noop*, *Anticipatory* and *Deadline*, some of them promising some guarantees. [C26] More information in the Linux documentation.
[2] This is why a performance requirement would never look like "The system must be able to handle a message in less than 50 ms." but rather like "The system must be able to handle 20 messages per second." (see chapter 2.3 Requirements).

With virtualization, several *virtual machines* (isolated from each other) can be simulated by one real one. Even if the real one may have to be more powerful, it may allow:

- a better use of the resources (50 % of the processor may be used on average instead of just 20 %, for example);
- an economy of space and energy;
- a lower purchase cost.

These are the reasons for which Comsoft wants to use virtualization.

Note: Server consolidation may also bring performance improvements, contrary to what one might think. For example, two applications may communicate together faster if they are running on the same physical hardware platform than if they have to communicate over a network.

Software development

A developer may want to develop an application that could damage his/her development environment. This developer could then safely test his/her application on a virtual machine, as isolated as a real machine.

He/She may also be developing an application for another hardware platform, which may not be available to him/her. This platform can then be emulated on the development platform.

Legacy software

Some applications may need an old platform architecture (which may moreover not be available on the market) or an old operating system to run. In the first case, the old platform can be emulated by a recent computer. In the second case, a special virtual machine can be used to run the old operating system, so that it isn't necessary to use a dedicated platform.

Load balancing and disaster recovery

An *image* or *snapshot* of a virtual machine can be made and moved to another physical machine, thus allowing easy migration of a virtual machine over physical ones. It is useful for load balancing and disaster recovery.

Virtual desktops

Virtualization used on workstations allows for example to switch in a few milliseconds from a Linux desktop to a Windows desktop.

High availability and clustering

Creating one virtual machine on *several* physical machines is one type of virtualization too. There are several models:

- Several physical machines simulate the same virtual machine in parallel in a redundant manner, thus improving availability (if one physical machine fails, the virtual machine still remains operational).

- Several physical machines simulate one more powerful virtual machine with a lot of resources.
- Hybrid model: combination of both previous models.

2.1.2.2 Virtualization Techniques

Virtualization is a wide domain regrouping several concepts and techniques. However specialists don't seem unanimous concerning the names given to these notions. We present them in this chapter with the most explicit names in our opinion. We will stick to these names in the whole document.

Emulation

This technique is the one best corresponding to the original concept of virtualization. An emulation is a **fully software reproduction** of a machine. Every hardware component (processor, memory and I/O devices) of the *emulated machine* is simulated. The main characteristic of emulation in comparison to other techniques is actually the software reproduction of the processor.

The software responsible for the emulation (i.e. the imitation/reproduction of a machine) is called an *emulator*. The emulator is generally an application run by the operating system of the *host* machine (the *host* operating system).

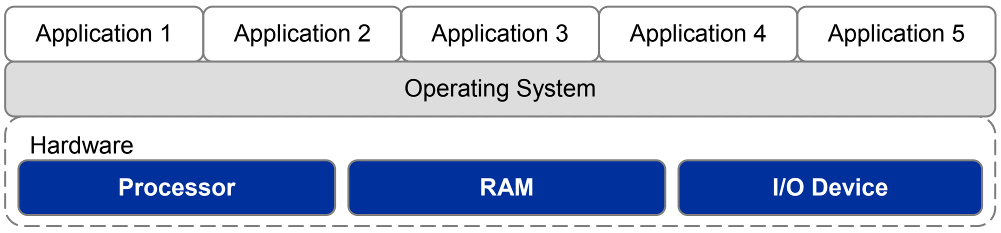


Figure 2-4: Structure of a classical computer running applications

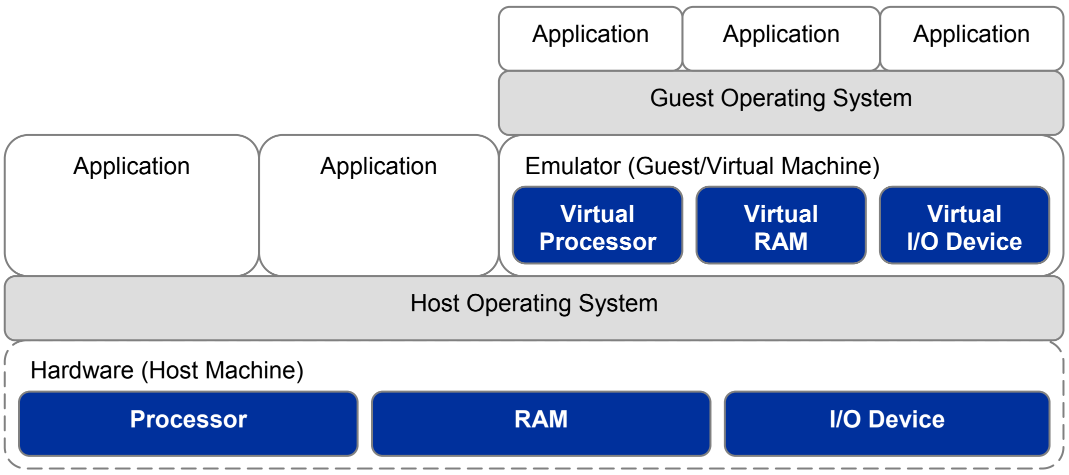


Figure 2-5: Computer running an emulator

Several emulators can run at the same time. The main component of an emulator is a command interpreter to replace the processor:

- Each instruction is executed by the interpreter whose implementation looks like the implementation of a normal language interpreter.
- Data intended to be sent to a screen can be redirected by the interpreter to a window or a terminal.
- Interrupts can be simulated by the interpreter by directly jumping to the first instruction of the guest operation system's handler corresponding to that interrupt request.
- Data from the keyboard is forwarded to the guest like any other interrupt.

Pros:

- The guest's software part (operating system and applications) doesn't need any modification. We usually say that it is not aware of being executed by an emulator instead of a real machine.
- It is possible to emulate a machine having an architecture very different from the host one. (For example PearPC emulates a PowerPC platform for the x86 architecture [V45].) Thus you can emulate an old computer which is no longer produced and continue to use applications designed for that machine.
- The host and the guest operating systems can be different.
- Isolation: The guest machine being usually seen as a normal application for the host operating system, a crash of the guest should not affect the host nor the other applications (and thus other emulators) running on the guest.

Cons:

- Overhead: since each guest instruction is interpreted, a guest being emulated by a host of the same architecture (e.g. an Intel x86 being emulated by a real Intel x86, in order to run several operating systems) can be about 5 to 10 times slower than a real machine, even for the most powerful emulators, because of the mean number of host instructions needed to emulate one guest instruction.
- Scheduling of several emulators running on the same host in order to guarantee respect of deadlines has to be made using the features offered by the host operating system to schedule standard applications. However some operating systems don't offer many scheduling possibilities.

Full Virtualization

The major inconvenient of the emulation is the huge overhead. Because of it, this solution is not adapted for server consolidation (see chapter 2.1.2.1 Reasons).

Full virtualization is the creation of virtual machines of the same type/architecture as the host machine. Since the *instruction set architecture*[1] is the same, there is no need to emulate each of the guest's instructions. The guest can directly use the available hardware, in particular the microprocessor(s), thus drastically reducing the overhead.

[1] The instruction set architecture is the hardware interface accessible/visible for the software. It encompasses for example the operations, the addressing modes, the visible registers, etc. You can find more about this topic in *Computer Architecture* [C4] and *Mikrocontroller und Mikroprozessoren* [C5].

Of course, this technique doesn't make any sense if there is only one guest. Why would you create one virtual machine identical to your physical one? It makes sense if you create *several* virtual machines on top of one physical one. You could then let different operating systems run on the same physical machine at the same time. It makes a server consolidation possible.

However several operating systems are now competing for the same resources. That is why we need a referee to supervise and allocate those hardware resources: this referee is named *Hypervisor* or *Virtual Machine Manager* (VMM).

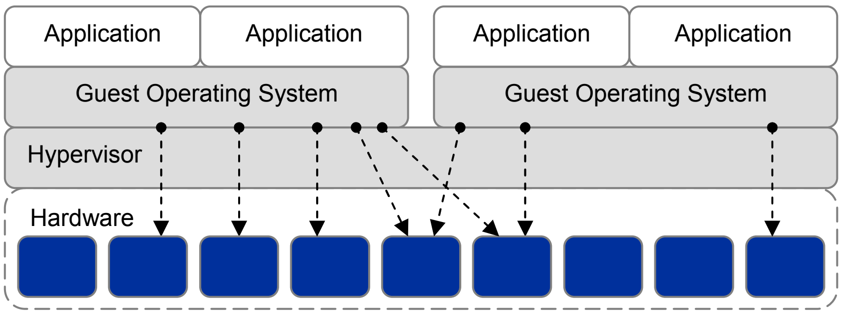


Figure 2-6: Full virtualization

Appendix A Implementing full virtualization shows how complicated the implementation of a full virtualization product can be and gives an overview of the different solutions.

Full virtualization enables the creation of virtual machines on top of one physical/real machine, where the virtual machines have the same *instruction set architecture* as the real one.

Pros:

- The guest's software part (operating system and applications) doesn't need any modification. (Same advantage as for emulation)
- The operating system of each virtual machine can be different.
- Isolation: a crash of a guest should neither affect the hypervisor nor the other guests. (Same advantage as for emulation)
- Lower overhead than for the emulation.
- Scheduling: the hypervisor may offer some reliable options to schedule the different virtual machines, since it is operating at a very low level.

Cons:

- Virtual machines must have the same architecture as the real machine.
- Even if the overhead is lower than for the emulation, it is still not negligible (about 30% [V6]).

Paravirtualization

With full virtualization, the goal is to reproduce exactly the behaviour of a real machine so that an unmodified guest operating system can run on the virtual machine without any modification and without even "knowing" that the underlying machine is virtual.

With paravirtualization, a virtual machine doesn't have exactly the same behaviour as a real one, so that the guest operating system has to be modified to be able to run. There are several reasons to this approach:

- As explained in appendix A Implementing full virtualization, it can be quite difficult to develop a hypervisor without hardware support. And the result, using dynamic recompilation, might not be as efficient as expected. Instead of converting each of the guest's instructions on the fly, the hypervisor performing paravirtualization lets all the guest's instructions be executed, considering that they are safe and that a guest needing to access hardware will voluntary "call" the hypervisor (by means of a *system call*, in this case a *hypercall*).

  But a hypervisor can't actually just assume that every guest is safe, otherwise a malicious or buggy guest could directly access a hardware resource which was assigned to another guest and crash it or even the entire system. One common solution on x86 architecture is to let every guest operating system run in protection ring 1, 2 or 3 and put the hypervisor and hardware resources in ring 0 (see appendix A Implementing full virtualization for more details). Of course those guest operating systems have to be modified to be able to run in a restrictive ring.

  So the use of the protection ring 1 was not a valid solution for full virtualization, but it becomes valid when combined with modification of the guest operating system.

- *Paravirtualization* can be used to make the whole system even more efficient. For instance, every modern hypervisor gives the possibility to create *virtual NICs* (Network Interface Cards) to make an Ethernet connection between the guests (each guest sees then an NIC, which is actually a virtual one set up by the hypervisor). Without any modification, a normal guest operating system would perform *cyclic redundancy check* (CRC) on data coming from each NIC to ensure data integrity, including on data coming from the *virtual NIC*, since the guest doesn't know that it is virtual. But testing data integrity without *telecommunication* is unnecessary and thus a waste of CPU time.

  In paravirtualization, guest operating systems (kernel + drivers) are modified to recognise *virtual devices* and benefit from this information. They can also be modified so that they don't perform some verifications or operations which are now already performed by the hypervisor.

Some specialists refer to *paravirtualization* as a subset of *full virtualization* or a solution to achieve it, whereas some others differentiate them more strongly (we share this second opinion):

- Full virtualization is the sharing of a real machine among virtual machines **with exactly the same architecture**, able to run operating systems originally designed for the real machine.

- Paravirtualization is the sharing of a real machine among virtual machines with the same architecture but **presenting a slightly different behaviour**, needing operating systems to be modified to be able to run.

By extension, the modification of a guest operating system to improve performance, even if it is not necessary to achieve virtualization, can be named *paravirtualization*. The modification is then often available as a package to install in the guest and it is generally recommended to install it. [VV22]

Note: Paravirtualization implies modifications in the guest operating system but **no modification in the guest applications**.

You can find more information on paravirtualization:

- *Xen and the Art of Virtualization* [VX2];
- *Computer Architecture* [C4];
- *Das Virtualisierungs-Buch* [V1].

The main drawbacks of paravirtualization are:

- It is not possible to modify every operating system (for example some operating systems are not maintained anymore and/or not open source);
- Thus the list of supported operating systems is shorter for a hypervisor using paravirtualization than for one using full virtualization;
- When modifying an operating system kernel, there is a risk of inserting bugs.

Operating System-Level Virtualization

"Operating system-level virtualization enables multiple isolated execution environments within a single operating system kernel." [VZ1]

These execution environments are often called *containers*, *virtual environments* (VE), *virtual private servers* (VPS), *partitions* or *jails*. They are isolated, which means:

- each container has its own set of processes, users, groups, etc.;
- one element inside a container can't see elements from another container;
- all containers share the same operating system kernel, which means they share the resources of the machine through the single kernel instance (responsible for allocating resources not only to processes, but also to containers), but not the entire operating system (e.g. for Linux, containers can have different distributions based on the same kernel);
- containers shouldn't be able to affect each other; they can only communicate through tools set up by the kernel for this purpose.

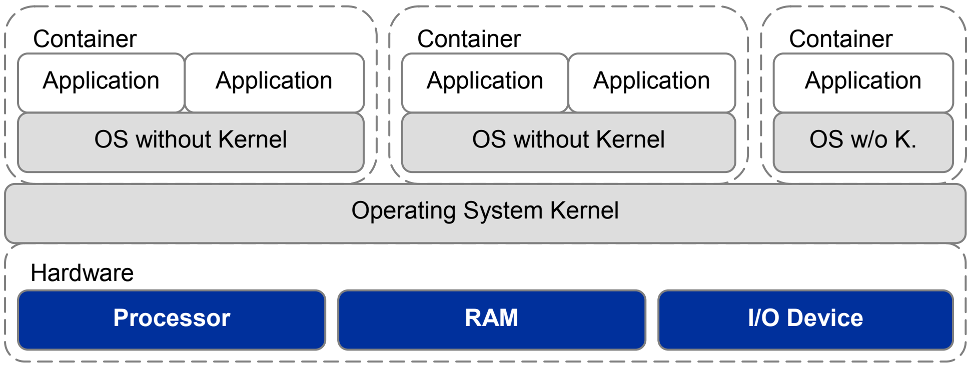


Figure 2-7: Operating system-level virtualization

Pros:

- Less overhead (only around 1-2% [VZ2], negligible in many scenarios) than with a classical virtualization (full virtualization or paravirtualization), because: [V7]
  - If the kernel of the OS-level virtualization solution is able to share the code of an application when several instances of this application are running (e.g. with a *copy-*

    on-write strategy), it will be able to do so even if the instances are running in different containers, achieving a "factorization" (suppression of identical elements), thus reducing the number of cache misses. Such factorization is hardly possible with a classical virtualization.
    - With OS-level virtualization, only one kernel runs, in opposition to classical virtualization. More CPU cycles remain available for the applications.
    - When a classical hypervisor performs a *virtual machine switch* (stops the execution of one virtual machine to give the processor to another one, according to the scheduling policy), the TLB (*translation look-aside buffer*, a cache for the page table entries) has to be flushed, leading to a lot of cache misses for the next instructions. This effect can be reduced by means of a *shadow page table.* [C4]
    - As shown in part Paravirtualization, some classical virtualization solutions can suffer from the fact that some operations/verifications are done twice (one time by the guest kernel, one time by the hypervisor). This drawback doesn't exist with OS-level virtualization.

- Flexibility: resource allocation between containers can be easily and dynamically changed. With a classical virtualization it isn't so easy: a classical guest kernel will crash if you reduce its amount of RAM (it would be the equivalent of removing a RAM module from a real computer while it's running). With paravirtualization though it's possible (e.g. with *ballooning*, a technique where the hypervisor asks a guest to free some memory [VX5]).

Cons:

- Less isolation: a bug in the kernel may crash the whole system, since all "guests" share the same kernel.
- No possibility of running several operating systems on the same physical machine.
- A kernel modification may be needed to support this virtualization feature, thus taking the risk of inserting bugs.

More information on this topic can be found in:

- *Das Virtualisierungs-Buch* [V1];
- *Virtualization in Linux* [VZ1];
- *OpenVZ User's Guide* [VZ2];
- *Performance Evaluation of Virtualization Technologies for Server Consolidation* [V7].

Operating System API Emulation

When only one architecture is involved (e.g. x86), if you want to run on one operating system (e.g. Linux) an application designed to be run on another one (e.g. Windows), a classical virtualization is not absolutely needed. The only reason why a Windows application cannot run on Linux, even if you use the same architecture in both cases, is because the interface offered by Windows to applications is different from the one offered by Linux. [V1]

An *OS API emulator* can for instance reproduce in Linux the interface of Windows, thus allowing a Windows application to run on Linux. This is the goal of the Wine project. [V44]

This type of virtualization has nothing to do with *resource allocation* and is therefore out of the scope of this study.

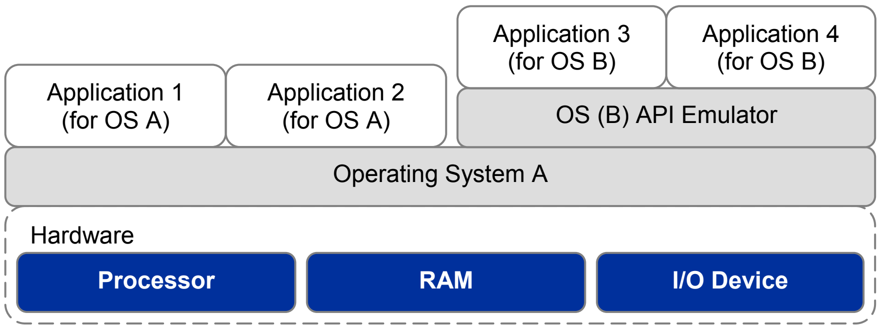


Figure 2-8: Operating system API emulation

Kernel in user mode

This technique consists in modifying the guest kernel (in a similar way to paravirtualization) so that it can be able to run as a normal application (in the protection ring 3) on the host operating system. [V1]

The major problem of this approach is that guest applications and guest kernel are running on the same privilege level, thus removing the guest kernel's protection against malicious or buggy guest applications. For this reason, this technique can't be used in production and is out of the scope of this study.

The main advantage of this approach is that the guest kernel can be traced and debugged using tools used to debug classical applications. For this reason, this technique is very useful in kernel development.

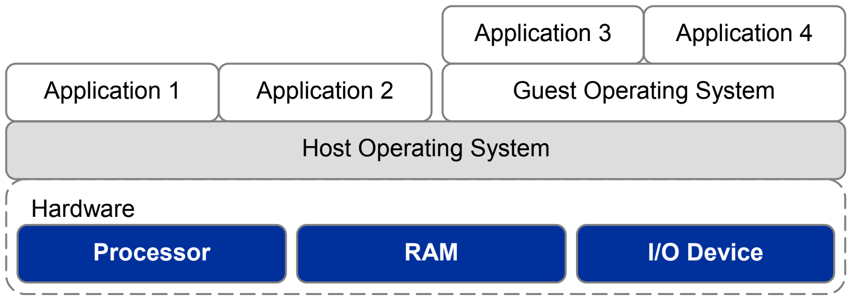


Figure 2-9: Kernel in user mode

Conclusion

Our project needs a solution for server consolidation enabling a precise configuration of the resource allocation and high performance. As already demonstrated, *emulation*, *OS API Emulation* and *Kernel in user mode* are not fitted for server consolidation.

OS-level virtualization should offer the best performances and the fact that all *containers* have to share the same kernel is not a problem for our project. However since the virtualization is done on a higher level than with classical virtualization, the timings for resources' allocation between the containers may not be precise.

Hardware-supported virtualization seems to be the solution offering the most predictable results, and therefore should be the best solution when dealing with real-time constraints. Moreover the hardware platforms used for this project can make use of this feature. However at this point it isn't clear if this technique is more or less efficient than virtualization based on *dynamic recompilation* (see appendix A Implementing full virtualization).

Paravirtualization may also be an efficient solution, either as complete virtualization technique, or as an optimization for one of the previously mentioned techniques.

2.1.3 Real Time

A non real-time system is a system (computing outputs from its inputs) where only the correctness of the result matters. In a real-time system the respect of time constraints is as important as the correctness of the result. [C7] [CR1] In such a system, a result delivered too late is as good as a wrong one.

Real-time systems are usually divided in two categories: *hard real-time systems* and *soft real-time systems*.

2.1.3.1 Hard Real Time

In such system (not necessarily a computer), the time constraints *must* be respected. For example an *airbag* (car safety device) is controlled by a hard real-time system. It must be guaranteed that the airbag will be deployed at the right moment (actually a time interval) after an impact. The deployment of the airbag is the result delivered by the system. If this result is delivered too late (or sometimes too early), it can be fatal.

Moreover, a system respecting the time constraints in 99% of all cases *is not* a hard real-time system. [CR1] A hard real-time one *always* respects them. Thus you cannot use a personal computer with a classical operating system to develop a hard real-time system. What would happen if the computer is slowed down by an antivirus update when you have a collision?

In a hard real-time system, the *worst-case execution time* (WCET) is very important [CR1]. It is the longest time needed by the system to deliver the result. A short *mean execution time* doesn't bring anything, if the WCET is too long to meet the deadline. That's why some computer optimizations like *cache memory* aren't used in hard real-time systems, since they only improve the *best-case execution time* (BCET) (and thus the *mean execution time* too), but not the WCET.

Such a system must be as simple as possible to be predictable. If it isn't predictable, you can't determine the WCET. Thus you can't prove that the system respects the time constraints.

The *predictability* is actually so important, that the difference between the WCET and the BCET (representing the *jitter*) should be as small as possible. [CR1]

Most complex real-time systems are driven by a software application made of several tasks or processes. The typical structure of such a task looks like this:

```
// initialization

// main loop:
bool terminate = false;
while (!terminate)
{
     event e = nextEvent(); // wait for an event
     switch (e)
     {
          case EVENT_TYPE_1:
               // real-time reaction to this event
               break;
```

```
            case EVENT_TYPE_2:
                 // real-time reaction to this event
                 break;

            // ...

            case QUIT:
                 terminate = true;
                 break;
       }
}

// termination
```

A *real-time operating system* is then responsible of scheduling the different tasks according to their *priority*, and offering them a basic set of objects and concepts to communicate (e.g. events, mutexes and messages) among them and with the outside.

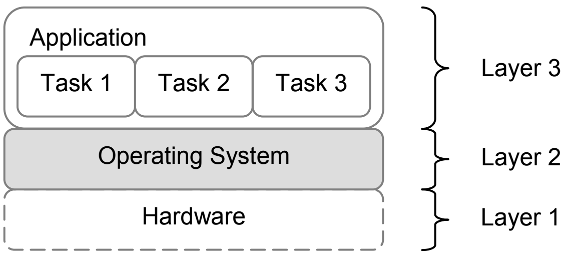


Figure 2-10: Real-time system

Let *p* be a function representing the predictability of a layer and delivering a value between 0 (unpredictable) and 1 (fully predictable). The predictability of the whole system is then the product of the predictability of each layer:

$$p_{system} = \prod_{i \in system} p_i$$

The system on figure 2-10 is a *hard real-time system* if:

$$p_{system} = p_{hardware} \times p_{OS} \times p_{application} = 1$$
$$\Rightarrow p_{hardware} = 1 \text{ and } p_{OS} = 1 \text{ and } p_{application} = 1$$

That is to say: the only way to manage to do a hard-real time system is to make every layer completely predictable.

2.1.3.2 Soft Real Time

A soft real-time system is a system where:

$$p_{system} \simeq 1$$

The time needed by the system to react or to do a certain work doesn't have to be totally predictable. The time constraints in such system should be seen as directives or guidelines. [CR1] Those time constraints can be transgressed as long as it doesn't happen too often. Usually the following is tolerated:

- A time constraint can be often transgressed if the transgression is limited in a tolerance interval.

- The transgression of a time constraint can be high if it happens rarely.

However the second assertion doesn't apply to Comsoft products, because as already explained in chapter 2.1.1 Comsoft products a high latency when accessing the hard disk drive is not acceptable.

The evaluation of a soft real-time system can often be carried out with only a general knowledge of the system and by collecting statistics on its behaviour, whereas a hard real-time system can only be deemed *hard real-time* after a very close examination of the implementation of each component (never after a statistical study).

2.2 Related Work

2.2.1 Existing virtualization products

The particularity of our approach is due to the fact that the company Comsoft needs a solution as quickly as possible. Therefore we renounced to design an adapted hypervisor or even extensions for an existing one. Instead we defined a series of indicators and implement tests carrying them out, in order to evaluate *existing* off-the-shelf solutions.

First of all several papers describe the state of the art virtualization, as does chapter 2.1.2 Virtualization. Jeff Daniels [V10] and Simon Crosby et al. [V16] present the concept of virtualization, its history, the different techniques to achieve it and the reasons to use it. He presents virtualization as a useful and mature technology. Mark F. Mergen et al. [V11] present the benefits of virtualization: flexible OS variety, productivity, performance, reliability, availability, security and simplicity. However Liana Fong et al. [V12] show that even if virtualization may provide simplicity, it also may add complexity at some levels. Ulrich Drepper [V21] describes all low-level performance problems occurring with virtualization.

Edward Ray et al. [V38] describe the security risks related to virtualization. They remind us that the isolation between two virtual machines is not as good as the one between two hardware platforms and highlight several security holes in hypervisors (e.g. *man-in-the-middle* attack during a *live migration*, *rootkits*, etc.). They present a set of best-practices to limit the risks.

Some articles focus on one virtualization product and help to establish a list of already existing products and to get a first opinion. Irfan Habib [VK2] presents KVM, Paul Barham et al. [VX2] Xen, Kirill Kolyshkin [VZ1] OpenVZ. We discuss their properties in chapter 3.4 Comparison.

Several documents [V13] [V14] [V15] compare briefly following virtualization products: VMware products, VirtualBox and KVM. However this comparison is made in the context of *desktop virtualization*, which is not directly related to our topic. Nevertheless those documents may be of interest since they help to get a first opinion on those products and to discover their features.

Rusty Russel presents in a paper [V36] *virtio*, an API and series of drivers for device paravirtualization in a guest Linux.

2.2.2 Evaluation techniques

Two main approaches are usually deployed to evaluate the performance of virtualization products, or more generally of hardware platforms and operating systems: micro-benchmarks and macro-benchmarks. Micro-benchmarks are series of small programs, each of them stressing one element or type of action (e.g. arithmetic operations, system calls, etc.) of the system and measuring its performance and properties. Macro-benchmarks are set of real applications (e.g. web servers, compilers, databases, etc.) or programs simulating real applications in order to evaluate the performance under real workload.

In their work, Padma Apparao et al. [V17] define a representative macro-benchmark for server consolidation. The problem of such an approach is that only generic mean indicators can be measured, like *mean throughput* of different resources or *mean CPU utilization*. In the best case, such a study could determine some *mean latencies*. It's definitely not adapted to our problem, where the real-time dimension plays an important role. In our study, we rather need to determine *maximum latencies*, *throughput variations* (and thus *predictability*) and *isolation* between virtual machines. Pure performance (that is to say *mean throughput*) is only a secondary criterion, allowing to decide between two *valid* virtualization products (offering a good predictability and isolation). This kind of study (with macro-benchmarks) is only useful in a non real-time context (in this case *Information Technology*) to compare the general performance of two virtualization products.

Tsuyoshi Tanaka et al. [V19] made also a study with macro-benchmarks. This study led to the conclusion that applications with heavy disk I/O and low CPU utilization are good candidates for server consolidation concerning performance. This is an encouraging result since this profile of application matches with Comsoft applications.

Jeanna Neefe Matthews et al. [V18] have made an important study on quantifying the isolation between virtual machines for several products. Their approach is similar to ours (used in chapter 3.5 Evaluation) in terms of strategy: they measure the performance of some virtual machines when other virtual machines are running programs stressing one element of the system (e.g. CPU or hard disk). However they use a web server to measure the performance: they used the amount of HTTP requests which got a reply "in-time" as an indicator for the *isolation*. Our opinion is that this indicator is way too coarse-grained for our study. The authors argued that other indicators can be used, and this is the case in our study: the stressing programs and the programs measuring the indicators are the same, like in a micro-benchmark approach, and are similar to their stressing programs. Moreover, they chose to measure the performance of web servers located in misbehaving virtual machines too (virtual machines where stressing programs are running), and use these results to conclude, which is, from our point of view, quite misleading: the behaviour of an application disturbed by another one running in the **same** virtual machine has nothing to do with isolation **between** virtual machines. However their study shows a **perfect isolation for VMware Workstation** (full virtualization) and a good isolation for Xen (full virtualization). The isolation of OpenVZ (OS-level virtualization) is not so good, particularly concerning the communication over the network.

Keith Adams et al. [V6] use both micro-benchmarks and macro-benchmarks to compare software-supported (binary translation) and hardware-supported virtualization. Their micro-benchmarks are adapted to our study (we used similar ones) but they limited their work to *mean execution times*, which is way insufficient for a real-time problematic. They conclude that hardware-supported virtualization is less efficient than software-supported virtualization, but their study is four years old and hardware-support was at the very beginning of its existence.

In *Xen and the Art of Virtualization* [VX2], Paul Barham et al. measure the performance, the isolation and the scalability of Xen by means of micro-benchmarks and macro-benchmarks. Unfortunately they focused again on *mean execution times*.

Stephen Soltesz et al. [V20] have made a comparison between OS-level virtualization and full virtualization, using both micro-benchmarks and macro-benchmarks. They concentrated their work on isolation (*fault isolation*, *resource isolation* and *security isolation*) and mean performance. They concluded that for CPU-bound applications, both techniques are equivalent but OS-level virtualization is more efficient on I/O. The fault isolation should be better for full virtualization. Both techniques should be equivalent concerning resource and security isolation. They argue that **efficiency and isolation are partially conflicting**, and that a trade-off has to be found. In our project, isolation is clearly more important than efficiency. They showed near-native performance for VServer (OS-level virtualization product) whereas the performance was clearly worse for Xen (full virtualization). They also showed that VServer scales better with the number of virtual machines. The performance isolation is better with Xen. Again they limited their study to mean throughput measurements, which is insufficient for a real-time problem.

Robert Kaiser [V22] has written an important document on virtualization for (hard) real-time systems. He explains how virtualization increases the reaction time (time between the occurrence of an event and the reaction of an application to this event) and induces a jitter. He explains that **if this jitter is short in comparison to the deadlines** the concerned real-time applications have to meet, **virtualization is suitable for a real-time system** without negative effect.[1] In the other case, the author exposes the requirements for a real-time hypervisor and proposes a design.

Vineet Chadha et al. [V23] described an interesting method that could be used to evaluate virtualization products. Instead of running the product directly on a hardware platform, it can be run in a *platform simulator* (or *emulator*) allowing to finely configure micro-architectural details of the platform to be simulated. A lot of low-level events can be gathered by means of the simulator, which can be correlated with the high-level events gathered in a virtual machine.

2.2.3 Future evolutions

Several papers propose different designs to improve the performance of current virtualization products. They are not directly useful for this study, but show that a lot of effort is currently put in improving virtualization.

Yaozu Dong et al. [V24] proposed a standardization of I/O virtualization in order to achieve better performance and predictability. They implemented it for Xen Hypervisor and measured the improvement.

Kaushik Kumar Ram et al. [V25], Guangdeng Liao et al. [V28] and Jiuxing Liu et al. [V29] present mechanisms and optimizations to achieve 10 Gb/s on network interfaces. These optimizations are for example the use of *multi-queue NICs*, *grant reusing*, *cache-aware scheduling* and *dedicated CPU-core* for polled I/O. They all obtained satisfying results.

Hwanju Kim et al. [V26] address the problem of starvation of I/O-bound tasks running in the same virtual machine as a CPU-bound task. They propose a solution for the hypervisor's virtual machine scheduler to boost those tasks without sacrificing CPU-fairness. Our opinion is however that this kind of optimization is partially in contradiction with the principle of isolation: the minimal resource amount assigned to a virtual machine should be independent of its workload and of the workload of other virtual machines. This is actually a **danger that Comsoft has to take into account:** virtualization products are "benefiting" of more and more optimizations, which means that a virtualization product adapted to Comsoft's project may not be adapted anymore in a few years.

This problem of starvation of I/O-bound tasks had already been highlighted by Diego Ongaro et al. [V33] They studied the impact of the hypervisor's virtual machine scheduler on I/O performance.

Chuliang Weng et al. [V27] also propose an optimization for the hypervisor's virtual machine scheduler. Some virtual machines are running concurrent applications and the hypervisor should try as much as possible to schedule the virtual CPUs of this virtual machine on physical CPUs at the same time to improve performance, without violating CPU-fairness. A virtual machine could be declared as *concurrent* to benefit from this optimization.

[1] It seems to be the case for Comsoft systems, since this jitter should be of the same order of magnitude as a time-slice of the hypervisor's scheduler (usually 100 ms). The precision/granularity of time constraints imposed to Comsoft applications is one second.

Himanshu Raj et al. [V30] expose the concept of *self-virtualized I/O device*. Such device provides virtual interfaces to virtual machines. The goal is to increase throughput and scalability and to reduce latency, by reducing the involvement of the hypervisor in device virtualization.

Seetharami R. Seelam et al. [V35] designed and implemented a virtual I/O scheduler focusing on fairness between applications (or virtual machines) and isolation. However this scheduler does not take latency into account for the moment.

Jianyong Zhang et al. [V37] designed a scheduler for storage virtualization. This scheduler isolates throughputs and guarantees statistically a certain latency for a given throughput. *Statistically* means in this case that 95 % of the requests will be handled in time. As explained in chapter 3.5.1.1.5 Scheduling we defined in our project an indicator named $bound_{95\%}$ which is even stricter.

2.2.4 Related topics

Server virtualization may also benefit from works coming from related topics.

Philip M. Wells et al. [V31] wrote an interesting paper on *multicore virtualization*. The main idea is to let the hardware exposes virtual processors and map them to physical processors, instead of letting software (operating system or hypervisor) handle it. The result is a performance gain, as well as the possibility to handle heterogeneity due to for example changing reliability, power or thermal conditions.

Lan Huang et al. [V32] designed and implemented a system for storage virtualization named *Stonehenge*. In contrary to most storage virtualization systems, this one does not only focus on capacity but also on bandwidth and latency, being able to deliver real-time guarantees.

Gernot Heiser showed in a paper [V34] that current virtualization products are not suitable for embedded systems: the isolation principle actually doesn't fit the requirements of embedded systems: efficient sharing, priorities (how to give a priority to a virtual machine if all tasks of all virtual machines have overlapping priorities?), energy management, fine-grained encapsulation.

2.2.5 Other utilizations of virtualization

Several papers describe the use of virtualization to improve security. Even through this topic is not ours, those document are likely to bring useful information.

Igor Burdonov et al. [V39] present a design where two virtual machines are running on the same hardware platform:

- One *private* virtual machine is isolated from the outside world (e.g. internet). It executes trusted processes and contains sensitive data. Even if the guest operating system is untrusted, the isolation provided by the hypervisor makes the run processes trusted.
- One *public* virtual machine is connected to the internet. It can only communicate with the private virtual machine by means of a restricted set of RPCs (Remote Procedure Calls).

Yih Huang et al. [V40] present a framework for tracking application interactions while putting each application in a different virtual machine, using operating-system level virtualization.

Application tracking and monitoring can be made by running all programs in a virtual machine and the monitor in another one, in order to provide a good security. However with full virtualization, such a solution may create a significant overhead because of the frequency of *VM Entries* and *Exits*. Monirul Sharif et al. [V41] propose a new design, where the monitor is located in the same virtual machine as the system to be monitored, but in a separate hypervisor protected guest address space. Such a design has been made possible by hardware support for virtualization (e.g. Intel VT). Such a design offers the same level of isolation between the monitor and the monitored system, without overhead due to virtual machine switches (i.e. entries and exits) when an event triggers the monitor (only one virtual machine exists). They analysed the security of their design and implemented a prototype over KVM, whose evaluation showed very satisfying results.

Haibo Chen et al. [V42] propose a solution based on virtualization to apply patches and upgrades on an operating system kernel while it is running, i.e. without service interruption (*live update*). The main idea is to run the system over a hypervisor and let the guest operating system initiate the live update by means of a *hypercall*. Therefore the guest operating system will remain blocked until the end of the live update, guarantee the atomicity of this operation. If the update is short, no service interruption will be noticed. They designed the whole framework and implemented a prototype for Linux over Xen. The measured overhead is negligible.

Youssef Laarouchi et al. [V43] explore the possibility to run the same application on different operating systems at the same time, so that in case one operating system is corrupted, leading to a wrong output from one instance of the application, this problem will be detected by comparison of the outputs. This improves the dependability of the application.

2.2.6 Particularities of our work

Only a few scientific studies about server virtualization and real-time have been done for the moment. Apart from the fact that our work is applied to a very concrete problem coming from an industrial company, its scientific part (evaluation of existing virtualization products) is singular. To our knowledge, this work is the first one with following particularities:

- It focuses not only on mean performance but also on **predictability** and **real-time.** When measuring a throughput, the **deviation** of this throughput over the time is taken into account. We measured **maximum latencies** instead of mean latencies. **Isolation** has also been carefully evaluated.
- The **indicators** we defined to evaluate virtualization products are **much stricter** than those usually used for this purpose.
- Care has been taken in order to define precise indicators and **fine-grained tests** to measure them. All **precautions** have been **explained**.
- All tests are **documented**, **reproducible** and **easily portable**. They are designed to be **reused**.
- In particular a phenomenon similar to "**Heisenbugs**" has been taken into account when implementing evaluation tools: a computer system has a different behaviour while being under test or observation because it takes an active part in this observation. Firstly we reduced as much as possible this active part. Secondly we made sure that this distortion can only lead to worse results when evaluating a system.
- We **didn't focus on one particular** virtualized **resource**. We evaluated CPU, RAM, hard-disk, networking and graphical interface.

2.3 Requirements

Nomenclature:

Throughout this chapter the term

shall	specifies a requirement that is mandatory
should	specifies a requirement that is highly desirable.
may	specifies an optional requirement.

2.3.1 Technical Requirements

2.3.1.1 From Customers

The System Requirements Specification (SRS) for an AFTN/AMHS System [A4] describes the requirements for a system for the transmission, processing, and storage of aeronautical messages in connection with international AFTN and AMHS related networks and in compliance with the relevant ICAO recommendations. Only the following subset of these requirements may be compromised by the virtualization of the hardware and have to be taken into account during the next steps of the project. All other requirements from the document [A4] are unaffected by the project.

2.3.1.1.1 ATS Message Switching Centre

(1) The system **shall** be configurable to be able to exchange messages via AFTN, AMHS and a combination of both without discontinuities in the reception and transmission of messages.

(2) The system **shall** be capable to synchronise with external NTP time servers.

2.3.1.1.2 System Management

(1) The system **shall** implement continuous supervision of the health state of all system components.

(2) The system **shall** implement fault and error management and shall log all faults and errors in an appropriate log.

(3) This log **shall** comprise a configurable period of time (not less than one month).

(4) The system **shall** be able to automatically switchover or re-assign resources upon detection of a fault.

(5) A switchover or re-assignment **shall** not take longer than ten seconds.

(6) The system **shall** be able to perform automatic re-initialisation (e.g. reboot of the affected system components) upon detection of a fault.

(7) Pending message transactions including message queues **shall** be restored in less than five minutes.

(8) It **shall** be configurable whether a re-initialisation is performed or not and with or without traffic recovery.

(9) A complete re-initialisation (e.g. after power failure) **shall** not take longer than ten minutes.

(10) The system **shall** collect diagnostic and statistical information of the various system components.

2.3.1.1.3 Reliability, Maintainability and Availability

(1) The system **shall** be available 24 hours per day / 7 days per week.

(2) The total availability rate of the system **shall** be greater than 99.999%.

(3) The values indicated in the offer **shall** be based on the study of existing operational systems.

(4) The hardware **shall** be deployed in a redundant way that allows exchange of components of the operational system without interruption of service.

(5) There **shall** be no single point of failure in the system. Databases used for online traffic storage and configuration data **shall** be clustered without shared use of hardware and software modules. LAN and WAN adaptors **shall** be redundantly implemented in hot-standby configuration.

(6) The offer **shall** detail the procedures for recovering from failure situations in terms of rapid system re-installation.

(7) The offer **shall** detail the Mean Time Between Failures (MTBF) and the Mean Time to Repair (MTTR) predictions for the system.

2.3.1.1.4 Performance and Sizing

(Those values are the ones used for "standard" customers. Some customers may have stricter requirements.)

(1) The system **shall** support an average message input rate of 20 messages per second with a mean input-output ratio of 1:2 with no accumulation of messages within the system.

(2) With a mixed message input of AFTN and AMHS/X.400 messages, the system **shall** support an average message input rate of at least 20 AFTN messages and at least 20 AMHS/X.400 messages.

(3) This means, with AFTN messages only, that the system **shall** support an average message input rate of 20 AFTN messages per second; with a mixed message input, the system **shall** support a sustained message input rate of 10 AFTN and 10 AMHS/X.400 messages per second.

(4) The system **shall** operate with the average message input rate with any combination of message addressees.

(5) The AFTN/AMHS gateway **shall** support the simultaneous transformation of 20 AFTN to AMHS messages per second and 20 AMHS to AFTN messages per second.

(6) The message transfer time within the system **shall** not exceed 1 second operating at the sustained message input rate.

(7) The system **shall** be able to process a peak load which is three times the average load detailed above for at least one hour without queuing of messages.

(8) The requirements above base on the following message profile:

- average message text size: 1.500 bytes;
- minimum message text size: 100 bytes;
- maximum message text size: 15.000 bytes.

(9) For the purpose of performance testing, the tenderer **shall** apply an appropriate message mixture in order to achieve the average, minimum, and maximum message text sizes above.

(10) The response time for operator commands **shall** be less than 500 ms under average load and less than 1000 ms under peak load conditions (excluding database retrievals).[1]

2.3.1.1.5 Redundant and Single System Operation

(1) The system **shall** be configurable to work as one redundant system or as two single systems (e.g. a single system for testing and a single system for training purposes).

(2) Each single system **shall** provide the full system functionality.

(3) The system **shall** be able to automatically and manually switchover or re-assign resources upon detection of a fault. A switchover or re-assignment shall not take longer than ten seconds. Pending message transactions shall not get lost.

(4) It **shall** be possible to split a redundant system into two single systems by means of configuration only.

(5) No hardware modifications **shall** be necessary for such splitting.

2.3.1.1.6 CADAS-ATS

(Those values are the ones used for "standard" customers. Some customers may have stricter requirements.)

(1) The AMHS UA Terminal System **shall** be able to run 50 simultaneous terminal sessions at a total load of 20 user message transactions per second.

2.3.1.1.7 Hardware Components and Power Supply

(1) The system **shall** consist of the following components:

- two AFTN/AMHS servers, operated in main/hot standby configuration; each of the servers shall be equipped with a redundant disk array (RAID 1 or RAID 5 configuration);
- [...]

(2) Uninterrupted power supply (UPS) with a capacity to supply the AMHS switch and its peripherals during power failure for one (1) hour **shall** be provided.

[1] This requirement originally concerns the *operator's working positions* (i.e. workstations, see chapter 2.1.1 Comsoft products) and thus not our study. We decided to apply it to the servers too and to keep it in mind for the evaluation of our implementation.

2.3.1.1.8 Other Requirements

(1) In addition to the compliance to the requirements as listed above the tenderer **shall** be prepared to demonstrate on request of the contractor its capabilities by a practical presentation including all major building blocks of its solution.

(2) The presentation **shall** prove the capabilities of the tenderer to cope with the requirements of this specification and to comply with them in their entirety.

2.3.1.2 From Comsoft

(1) The solution **shall** be based on servers of the HP ProLiant DL series, and **should** be based on the same servers as those already used to build solutions without virtualization, except for some minor configuration details: e.g. the servers **may** have more RAM as usual.

(2) No unusual hardware component **shall** be needed.

(3) The *guest* operating system (see chapter 2.1.2 Virtualization) **shall** be a Red Hat or a Fedora distribution, and **should** be the same as the one used for a solution without virtualization.

(4) The *guest* operating system kernel **shall** not be recompiled but the default kernel of the distribution **shall** be used, in order to benefit from the safety assessment of the distribution.

(5) Existing software components **shall** not need any modification to be compatible with the new solution, except configuration components.

(6) New software components needed for this project **shall** benefit from an extensive support, either from a significant company or from a large community.

2.3.2 Commercial Requirements

(1) The solution **should** generate savings on purchase in comparison to a classical solution. If not, the investment **shall** be reasonable, so that savings can be expected thanks to a lower energy consumption on the long view.

(2) The solution **may** generate savings concerning the assistance and maintenance services from HP.

2.3.3 SWAL 3

SWAL 3 stands for *Software Assurance Level 3* and is one aspect of the *Recommendations for Air Navigation Systems Software.* [A3] These recommendations imply following requirements:

(1) This document **shall** demonstrate the fulfilment of each mentioned requirement: a traceability between each requirement and each justification **shall** exist.

(2) The version of each software element of the final implementation **shall** be mentioned.

3 Design

In the course of this study, we focus on the systems AIDA-NG, CADAS-ATS and CADAS-IMS (see chapter 2.1.1 Comsoft products).

3.1 Existing design

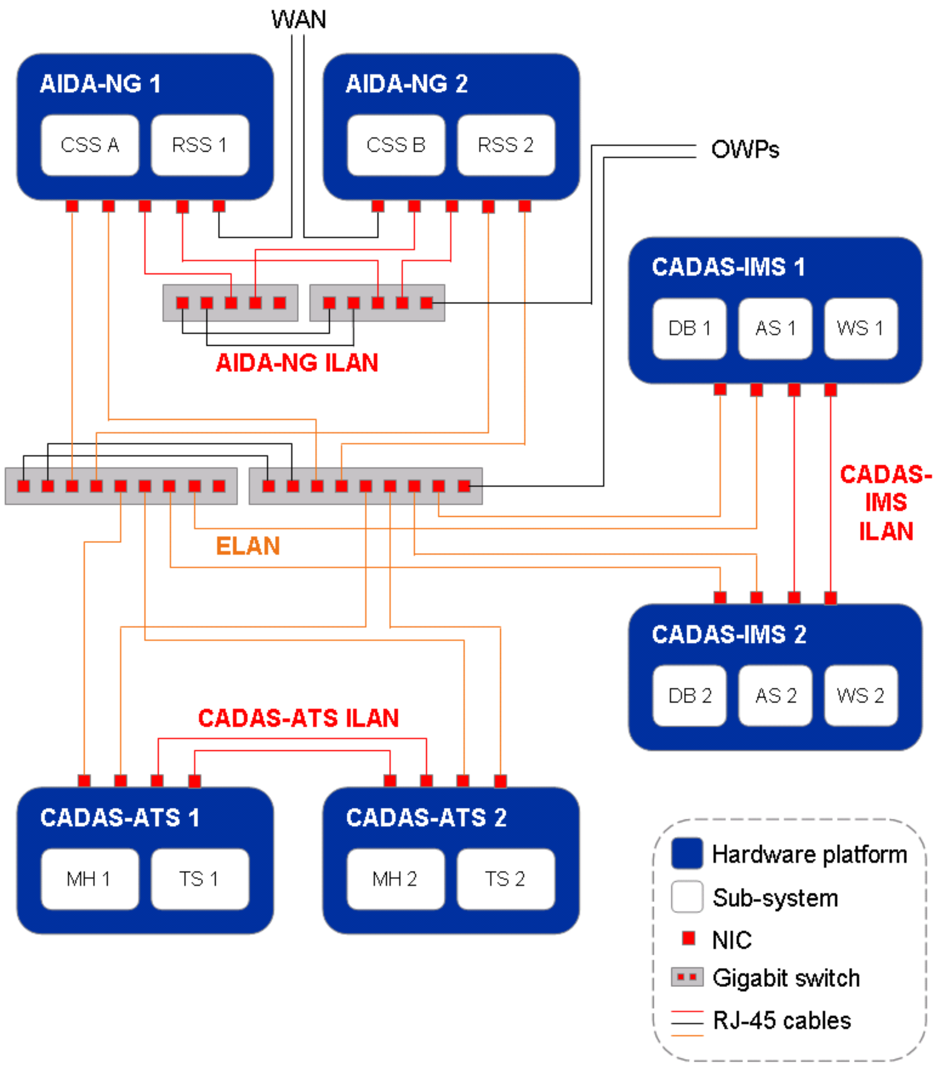


Figure 3-1: Architecture of a standard Comsoft solution

The architecture of a standard Comsoft solution based on those three systems/products looks like the figure 3-1. It may seem complicated at first but it's actually quite simple:

- each system runs on two different hardware platforms, to ensure redundancy;
- each system runs on its own set of hardware platforms, in order to guarantee that the performance requirements are fulfilled (see chapter 1.1 Motivation). The goal of this study is to reduce the number of hardware platforms.
- All hardware platforms communicate together by means of the ELAN (see chapter 2.1.1 Comsoft products). To ensure redundancy the ELAN is composed of two

switches connected to each other[1] and every hardware platform is directly connected to both switches (that is why there are 12 orange cables).

3.2 New design

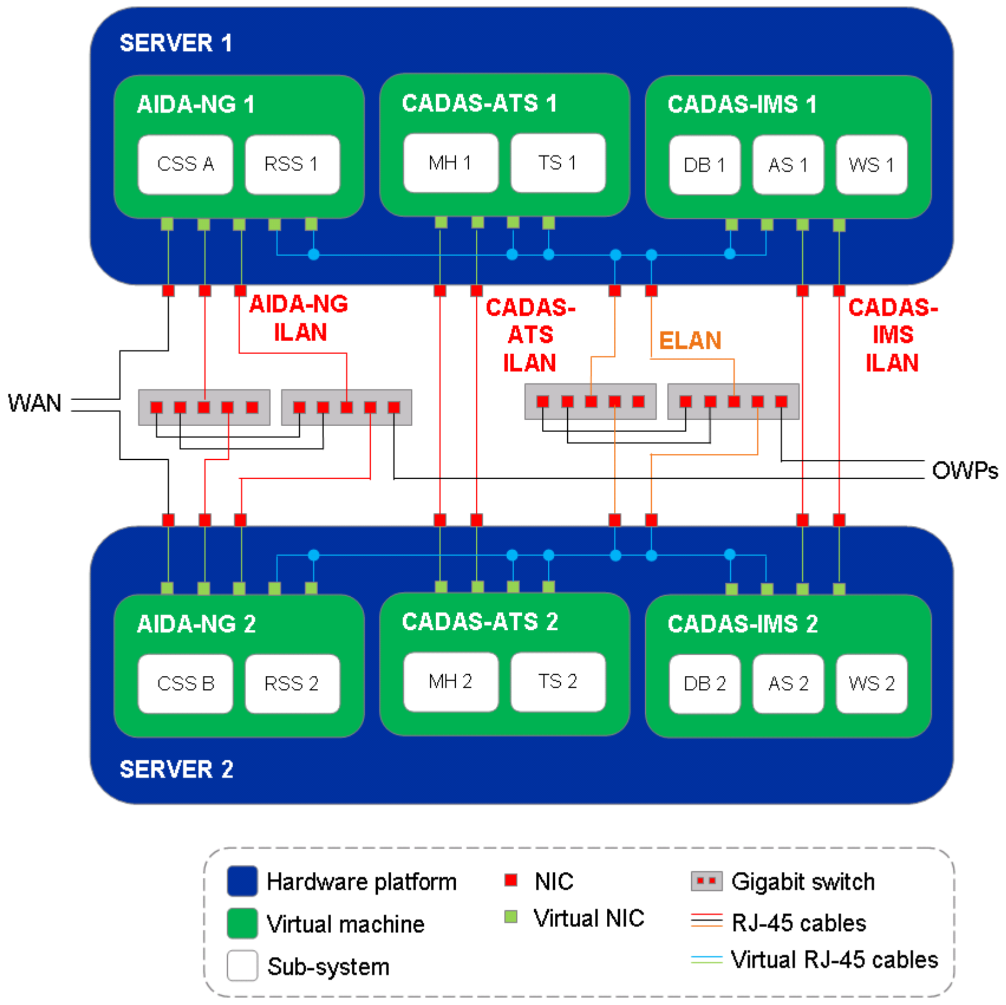


Figure 3-2: Design of the new solution

Figure 3-2 shows the design of the new solution. This design is optimal because it has the minimal number of hardware platforms (two is the minimum in order to ensure hardware redundancy) and still fulfils following constraints:

- the two instances of each sub-system are running on different hardware platforms;
- all physical components of the ELAN and ILANs are redundant, like in the classical solution;
- the influence of a system on another is limited by the existence of virtual machines and by the virtualization layer (the ability of virtualization products to create this isolation is evaluated in chapter 3.5 Evaluation).

[1] The link between both switches is redundant. The switches have to be configured so that this redundant link doesn't create a loop in the network.

As you can see in figure 3-2, the virtualization must have a mechanism to virtually connect the virtual network interfaces to the physical ones. Please note that redundancy for the virtual network interfaces is pointless (if the virtualization layer crashes, all virtual NICs crash) and is not needed by the software part: in a classical solution, *bonding* interfaces in *hot standby mode*[1] are created, so that the software part doesn't even "know" that all interfaces are doubled (for example the software part of AIDA-NG only sees one network interface to the ELAN).

If the virtualization layer supports an equivalent to this *Linux bonding*, being able to connect one virtual interface to two physical ones, the number of virtual network interfaces can be reduced:

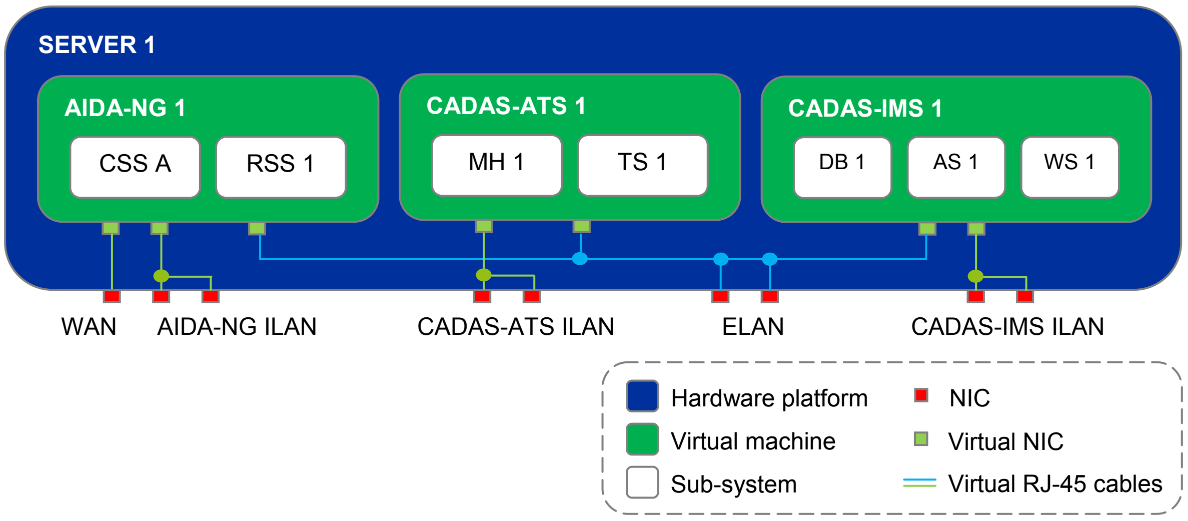


Figure 3-3: No redundancy for virtual NICs

Pros of letting the virtualization layer ensure the *bonding* functionality:

- simplification of the virtual cabling;
- the problem detection is done at a lower level: it may reduce the reaction time.

Con:

- the guest virtual machine is not aware that a physical link is down

From the previous design we can deduce that:

- three virtual machines will be created on each hardware platform;
- the maximum number of virtual NICs for one virtual machine is five;
- each hardware platform needs nine physical NICs.

[1] A Linux bonding interface is a logical interface based on several physical ones for transparent redundancy. *Hot standby* and *load balancing* modes are supported. [C21]

3.3 Compliance with the requirements

With this new design, following requirements are fulfilled:

Requirement	Summary	Justification of the compliance
2.3.1.1.1.(1)	System configurable to exchange aeronautical messages without interruption.	- Possibilities and configurability of the system unchanged. - Redundancy of all components.

2.3.1.1.3.(4)
2.3.1.1.3.(5)
	Redundant hardware and exchange of components without interruption of service. No single point of failure.	- Redundancy of all components.
		- One instance of each software component on each hardware platform.

2.3.1.1.5.(1)	System configurable to work as one redundant system or as two single systems.	- The system is composed of two hardware platforms which can be used separately. This change needs as much effort as without virtualization.
2.3.1.1.5.(2)	Each single system provides all functionalities.	- One instance of each software component on each hardware platform.

2.3.1.1.5.(4)
2.3.1.1.5.(5)
	Splitting the system doesn't require any physical modification (e.g. change to the cable layout)	- The system is composed of two hardware platforms which can be used separately. This change needs as much effort as without virtualization.
		- Some cables are now "virtual" and some parts of the cable layout can be changed virtually, which even extends the possibilities.

2.3.1.2.(2)	No unusual hardware component needed.	- See figures 3-1 and 3-2.

Table 3-1: Fulfilled requirements

Note: The compliance of the new design with some requirements is based on the fact that the current Comsoft design also fulfils these requirements.

3.4 Comparison of virtualization products

In this chapter, we compare different virtualization products in order to select those which are most likely to fulfil our requirements. Their evaluation is made in chapter 3.5 Evaluation.

3.4.1 VMware Infrastructure

VMware is a company with more than 7000 employees and 1.9 billion USD revenue in 2008, located in California and founded in 1998. [VV1] Its solution for server consolidation is VMware Infrastructure. [VV2] This solution is available in three editions:

- Foundation Edition includes the following products (description below):
  - the hypervisor ESX or ESXi (full virtualization + paravirtualization);
  - vCenter Server Agent;
  - Update Manager;
  - Consolidated Backup.
- Standard Edition includes:
  - the products already included in the Foundation Edition;
  - High Availability.
- Enterprise Edition includes:
  - the products already included in the Standard Edition;
  - VMotion and Storage VMotion (migration of virtual machines);
  - DRS (resource management) and DPM (power management).

ESX is a hypervisor based on Linux, providing several tools, whereas ESXi is a thinner hypervisor, not running on top of an operating system. Therefore ESXi offers better performance, security and reliability, and a smaller footprint due to a smaller code. [VV3] The tools provided by ESX can also be used with ESXi with the help of a special virtual machine named *vSphere Management Assistant*, available for free. [VV4]

Both hypervisors provide abstractions of the processor, the main memory, storage devices and networking, allowing to establish minimum, maximum and proportional resource shares for those resources. They require a 64-bit architecture[1] and use hardware-assisted virtualization (see appendix A Implementing full virtualization), supporting for example the Extended Page Tables from Intel VT-x. [C14] [V5] They support large memory pages, TCP Segmentation Offloading (TSO), TCP checksum offload[2] and Jumbo Frames. They can also make use of paravirtualization to improve performance of some devices. Paravirtualization for Linux guests is supported since Linux Kernel 2.6.21.

Note: Fedora 11 uses a 2.6.30 Linux Kernel.

[1] The current versions (ESX 4.0 and ESXi 4.0) require a 64-bit architecture. For 32-bit architectures ESX 3.5 and ESXi 3.5 can be used.

[2] TSO and TCP checksum offload let some operations (performed on TCP packets) be done by the network device itself instead of the processor. These techniques allow to reach a 1 or 10 Gb/s throughput with a low CPU usage.

ESX and ESXi support up to 1 TB of main memory and up to 255 GB can be assigned to each virtual machine. With the vSMP product, ESX and ESXi allow one virtual machine to run on top of up to eight physical processors[1] simultaneously. [VV5]

The VMDirectPath technique (a.k.a. Passthrough) allows to assign a physical hardware device (most PCI devices are supported) to one particular virtual machine so that it can directly access it. In such case VMotion can't be used.

Memory overcommitment, a technique allowing to assign more RAM to virtual machines than available (among other possibilities) [VX5], is also supported. With this technique, it is possible to allocate 12 GB (virtual) RAM to each virtual machine on a host that only has 12 GB. The advantage is that each virtual machine is able to use nearly the whole RAM, provided that the other virtual machines aren't using much memory at the time (otherwise the hypervisor has to *swap out* some virtual pages to the hard disk [C7]). Without memory overcommitment and with three virtual machines, you could only assign 4 GB RAM to each virtual machine, preventing them from using more, even if you can guarantee that they won't need more than 4 GB at the same time.[2]

Several powerful APIs (Application Programming Interfaces) are available to automate operations and manipulate the virtualization layer from a virtual machine or from the network: VIX API, vSphere CLI and vSphere PowerCLI. ESX and ESXi are also supported by *libvirt* (a standard open source project of virtualization API supported by Red Hat). [VK5]

VMware gives us to understand that ESX and ESXi are optimized for the Oracle database, which can be interesting for Comsoft, since some projects based on the product CADAS-IMS (see chapter 2.1.1 Comsoft products) use this database. [VV3] Moreover, HP and Intel recommend VMware[3]. [VV29] [VV30] Some VMware products can be bought directly from HP.

The additional products only available in the Standard Edition and the Enterprise Edition are not necessary for our projects: Comsoft has already implemented itself mechanisms for High Availability, and the ability to migrate running virtual machines from one platform to another is out of scope. VMware DRS (Distributed Resource Scheduler) has been designed for datacenters, in order to perform a dynamic load sharing over all available resources. It isn't needed here. VMware DPM (Distributed Power Management) is a module of DRS making the resource scheduler aware of energy consumption.

vCenter Server provides a centralized solution to manage several hosts. It isn't needed, because VMware vSphere Client is already provided with ESX and ESXi. vSphere Client is a Windows software allowing to manage a single host[4]. Several instances of vSphere Client can be run at the same time, so that one management computer can be used to configure several hosts at the same time. VMware Update Manager is a product for automating updates of VMware products. It isn't necessary because:

- The Comsoft policy is to deliver itself the software updates. Comsoft knows exactly the version of all software components deployed on all customer sites. Comsoft may want

[1] We are talking about eight *microprocessors*, not *cores* or *execution units*. See chapter 3.5 Evaluation.

[2] This technique also allows to allocate e.g. 64 GB to one virtual machine, even if the host only has 12 GB RAM. This strategy will make the hypervisor swap pages out to the hard disk instead of the guest operating system. It may be more effective, since the hypervisor is more aware of the underlying hardware than the guest. Another aspect of this technique is to allow virtual machines to share common/identical pages to free space in the main memory.

[3] Comsoft solutions are based on HP servers with an Intel architecture.

[4] Comsoft accepts to deliver a *management computer* running Microsoft Windows and dedicated to the administration of the hosts.

to integrate the deployment of VMware updates to its *Comsoft Configuration Management Suite* (see chapter 2.1.1 Comsoft products).

- In case Comsoft wants to use a VMware software to download and apply the latest patches for ESX and ESXi, vSphere Host Update Utility is delivered with ESX and ESXi and is highly sufficient.

VMware Consolidated Backup is a product allowing to backup virtual machines, with a lot of features. This is not needed for this project. Moreover, if a customer exceptionally wants to save a virtual machine, it can be done with vSphere Client.

In general, tools provided by the Foundation Edition are only needed to administrate a datacenter with a lot of hosts and guests. This is not the case in this project.

ESXi is available with a free license. When using this free license, ESXi is limited, so that it can't support migration features, high availability and all other features described as useless for this project. It's also limited to 4-way SMP (the hardware platform used by Comsoft has only two processors, see chapter 3.5 Evaluation), to 6 cores per processor (we only have 4 cores per processor) and to 256 GB RAM (we only have 12 GB). This **free version** provides vSphere Client and vSphere Host Update Utility, and **seems to be adapted to this project**.

Note: The APIs described before **should** be supported by the free version of ESXi. However this point is unclear and some of them may be blocked. We didn't find any categorical document on this point.

3.4.2 Other VMware products

VMware's other two main virtualization products are: [V1]

- **VMware Server**: this hypervisor is just an application running on top of a host operating system. This product is available for free and is presented by VMware as a "display case" so that companies can experience the advantages of virtualization before buying VMware Infrastructure (based on the more powerful ESX and ESXi). [VV6] [VV8] Therefore VMware Server shouldn't be used for this project.

- **VMware Workstation**: this product is designed to provide virtualization in a convenient way for the desktop. It isn't designed for server consolidation but to provide an easy way to run desktop applications on different operating systems on the same physical platform at the same time. Therefore VMware Workstation shouldn't be used for this project. [VV9]

3.4.3 Xen Hypervisor

Xen is originally a virtualization project started by Ian Pratt at the University of Cambridge and is now maintained by the company XenSource [VX3] [VX4], which has been acquired by Citrix Systems in 2007. Citrix is a company with more than 4 620 employees and 2.5 billion USD revenue in 2007, located in Florida and founded in 1989. Citrix focuses on virtualization products.

Xen Hypervisor (full virtualization + paravirtualization) is a **free** and open source hypervisor running on top of a Linux system (host operating system) [VX4], which makes it compatible with a lot of hardware (Xen thus inherits the compatibility of the host operating system) but probably generates more overhead than a thinner hypervisor like VMware ESXi. XenSource claims that Xen Hypervisor is "exceptionally lean with less than 150 000 lines of code" [VX1], but given that it's based on Linux and therefore enjoys a lot of kernel and driver code already written, we actually find that 150 000 lines are a lot.

It supports Intel VT-x and AMD-V to run unmodified guest operating systems with hardware-assisted virtualization since Xen 3.0. (See appendix A Implementing full virtualization) [VX4] [VX6] It can also run on other platforms without hardware support, making use of paravirtualization. (See chapter 2.1.2.2 Virtualization Techniques) Xen can run on both 32-bit and 64-bit architectures. Paravirtualization may lead to better performance but implies a modification of the guest operating system, which can be a violation of the requirement specifying that the default guest operating system's kernel has to be used. (See chapter 2.3.1.2 From Comsoft) Xen supports up to 64-way SMP and Intel PAE (Physical Address Extension) and therefore supports up to 64 GB RAM on a 32-bit platform. Memory overcommitment is not supported yet. [VX5]

Xen is supported by Intel [VX6] but seems to be abandoned by Red Hat[1], which now supports KVM. [VX10] Xen support was removed from Fedora since Fedora 9, but will probably come back in Fedora 13 [C18] with an additional tool to migrate a virtual machine from Xen to KVM.

Several CPU schedulers are available for Xen, some of them being soft real-time schedulers. [VX7] However a soft real-time scheduler is not absolutely needed on the virtualization layer, since the scheduling strategy we need is actually simple and is often achieved with a classical time-slice scheduling [C9]: each virtual machine should be guaranteed to get a minimal amount of the CPU time under all circumstances if it needs it.

On the networking side, Xen allows to limit the outgoing bandwidth of a guest, like ESXi. [V8] The support for TCP Segmentation Offload seems not to be completely stable and effective yet. [VX9] [VX8] Up to eight virtual network cards per virtual machine can be created since Xen 3.1 (the limitation was set to three in the previous versions), which is enough (see chapter 3 Design).

Xen has an equivalent of VMDirectPath (Passthrough) to assign a PCI device directly to a virtual machine. [VX8] The support of large pages is unclear.

Xen Hypervisor **seems to be adapted to this project.**

3.4.4 QEMU

QEMU is open source and can be used as a complete machine emulator or as a hypervisor (full virtualization, using hardware support if available or dynamic recompilation). [VK4] QEMU achieves better performance when used as a hypervisor, and this is the way KVM uses QEMU.

3.4.5 KVM

KVM (Kernel-based Virtual machine Monitor) is a **free** and open source virtualization project originally created by the company Qumranet but now owned by Red Hat. [VK3]

It's based on QEMU and needs hardware support (Intel VT-x or AMD-V). It runs on top of a host Linux operating system (it's a kernel module from the mainline Linux) but the strategy of KVM is to merge the hypervisor and the host operating system as much as possible to avoid superfluous operations. [VK6] Like Xen and VMware ESXi, it virtualizes the CPU, the main memory, the hard-disk drives and the network cards. KVM can also make use of

[1] Comsoft solutions are based on the Fedora distribution from Red Hat.

paravirtualization to improve performance for some devices (network, hard-disk, main memory, etc.), using Virtio. [VK3] Memory overcommitment is still under development.

Virtual processors appear to be normal threads in the host operating system, [VK6] so if all virtual machines have the same amount of virtual processors, they should get the same amount of physical CPU time. It's the only way to configure CPU utilization.

KVM allows the Passthrough technique on platforms equipped with the Intel VT-d technology. [VK7] Our platforms have this technology. [C14]

3.4.6 Linux VServer

Linux VServer is an open source project of operating system-level virtualization (a.k.a. *Partitioning*, see chapter 2.1.2.2 Virtualization Techniques). The project started in 2001. [VS1] Even though VServer is a well known project, it doesn't benefit from as much support as ESXi, Xen or KVM.

With operating system-level virtualization the host and the guest share the same kernel. Thus it's important that VServer can be used with Fedora and without the need to recompile the kernel, because of the requirements. (See chapter 2.3.1.2 From Comsoft) This is actually the case: VServer can be installed on Fedora by means of simple RPM packets. [VS2]

The overhead induced by this solution should be only about 1-2 %, because of its nature: operating system-level virtualization. Most resource usage can be limited, even if the way to do it sometimes seems a little complicated or vague. [VS1] Linux VServer relies entirely on the different standard Linux disk schedulers ("noop", "anticipatory", "deadline" and "cfq") and doesn't provide any additional way to limit disk I/O. It is unfortunately **not sufficient for this project** (see chapter 2.1.1 Comsoft products).

VServer doesn't provide the possibility to create virtual network devices. "Networking is Host Business" [VS1], which removes a lot of flexibility. It doesn't seem to provide any API and isn't supported by *libvirt*. [VK5]

3.4.7 OpenVZ

Like VServer, OpenVZ is an open source project of operating system-level virtualization. [VZ3] OpenVZ is the basis for Parallels Virtuozzo Containers, a commercial virtualization product. When Parallels started the Virtuozzo project, they had to bring modifications to the Linux kernel, with the obligation to make those modifications public (open source), because of the GPL license of the Linux kernel. OpenVZ is therefore the open source part of Virtuozzo. The first version was released in 2001. Virtuozzo is nowadays a common solution for web server virtualization, widely used by companies offering web hosting services.

OpenVZ is based on a special Linux kernel, which makes it **unadapted to this project**. [VZ3] However we can imagine that in a few years OpenVZ might appear as a simple collection of kernel modules. **Comsoft should keep a watch on this project**, because it seems very promising.

The overhead induced by this solution should be close to zero, like Linux VServer. Amount of CPU time can be limited for each container and is anyway assigned on a container basis (not on a thread basis). [VZ2] A special I/O scheduler has been introduced, on a container basis, guaranteeing that no container can saturate an I/O channel. [VZ4] The whole networking part of OpenVZ is much more oriented towards the configuration of each container than VServer, thus offering an advanced virtual networking functionality. The amount of main memory for each container can be limited. OpenVZ is supported by *libvirt.* [VK5]

3.4.8 Other virtualization products

Other more or less known virtualization products exist. *User Mode Linux* (kernel in user mode) is for example a special guest Linux kernel running as a normal application over a (host) Linux. [V1] (See chapter 2.1.2.2 Virtualization Techniques) Because the guest kernel isn't a standard Fedora kernel, this solution isn't adapted to the project.

Microsoft has virtualization products too. Its product for server consolidation is Virtual Server (the last version being 2005 R2 SP1). [VM1] This product wasn't retained, for several reasons:

- In the domain of air safety, a system whose main part relies on Microsoft Windows is often frowned upon. This argument may not seem impartial but the fact is: it can lead to marketing problems, even if the system is stable. And Microsoft Virtual Server runs on top of Microsoft Windows.
- It would have been worth evaluating this solution if some technical aspect made it particularly different from other solutions but that isn't the case.
- In the FAQ [VM2], Microsoft claims that some versions of Linux as guest operating system are supported but the page describing the product doesn't even mention it. [VM1]
- Some limitations like "the guest operating system must be 32-bit" or "the guest operating system can run only on one processor" (no virtual SMP) let us think that Virtual Server is not a mature product yet. The reason for this limitation is easy to guess: only the latest beta versions of Virtual Server support hardware assisted virtualization. [VM3]

An other product by Microsoft is Windows Virtual PC, which is a desktop virtualization tool. It doesn't support Linux as a guest operating system, but it *may* work.

The company VirtualLogix created a "real-time hypervisor" named VLX. [V46] But this product doesn't seem to benefit from extensive support and a large user group. Real Time Systems GmbH [V47] and KUKA [V48] developed their own real-time hypervisor too, but it doesn't seem to have sufficient support and user group either.

National Instruments developed a real-time hypervisor too, which may benefit from extensive support, but the only supported guest operating systems are Windows XP and LabVIEW Real-Time. [V49]

3.4.9 Conclusion

	ESXi	Xen	KVM	VServer	OpenVZ
Hardware support	√	√	√		
Device paravirtualization	√	√	√		
Thin virtualization layer	+	-	~	++	++
Configuration of CPU utilization	√	√	x	~	√
Configuration of main memory allocation	√	√	√	~	√
Configuration of hard-disk allocation	√	√	√	√	√
Virtual networking	√	√	√	x	√
Passthrough	√	√	√		
TCP Segmentation Offloading	√	~	?	√	√
Large Pages Support	√	~	?	√	√
Support at least 12 GB RAM	√	√	√	√	√
Memory overcommitment	√	x	x		
Virtualization API	√	√	√	x	√
Supported by a large community	++	++	++	~	+
Free	~	√	√	√	√
No compilation of the guest kernel	√	√	√	√	x
Quality and coverage of the documentation	+	~	~	-	~

Table 3-2: Comparison of different virtualization products

Only three products may be suited to our project: ESXi, Xen and KVM. Considering the time available for this study, we had to select two of them. **ESXi** seems to be the best candidate and was obviously selected. The choice between Xen and KVM was not easy but **KVM** was selected for two main reasons:

- The thin virtualization layer of KVM makes us think that we can reach more predictable results than with Xen.
- Comsoft insisted on evaluating KVM because of the recent abandon of Xen by Red Hat in favour of KVM (see chapter 3.4.3 Xen Hypervisor).

3.5 Evaluation

After having selected ESXi and KVM, this chapter shows the tests performed, in order to:

- verify and prove that those products fit our needs;
- find which one best fits our needs.

Two identical instances of the following hardware platform were used to perform the tests. This platform is an HP ProLiant DL380 Generation 6 [C11] with following characteristics:

- Processors: 2x Intel Xeon E5520: [C14] [C16]
  - 8 MB unified L3 cache (data + instructions)
  - 2.26 GHz
  - 64-bits
  - per processor: 4 cores / 8 threads
  - per core:
    - 32 KB L1 data cache
    - 32 KB L1 instruction cache
    - 256 KB L2 cache (data + instructions)
- RAM: 12 GB (6 x 2 GB) PC3-10600R (DDR3-1333) Registered DIMMs
- Two RAID Controllers: HP Smart Array P410i and HP Smart Array P410.[1] [C12] [C13]
  For each of them:
  - 512 MB Battery-Backed Write-Cache
  - RAID 0/1/1+0/5/5+0
  - 2 Channels/Ports (see figure 3-4)
- Hard disk drives: 16x HP 146GB 3G SAS 10K SFF DP ENT HDD
- A total of 12 Gigabit network interfaces:
  - Four integrated Gigabit network interfaces
  - Two PCI-Express cards (HP NC364T PCIE 4PT), for each of them:
    - Four Gigabit network interfaces.

[1] No document indicates the difference between P410i and P410 or even mentions that they are different. We therefore consider that they are identical.

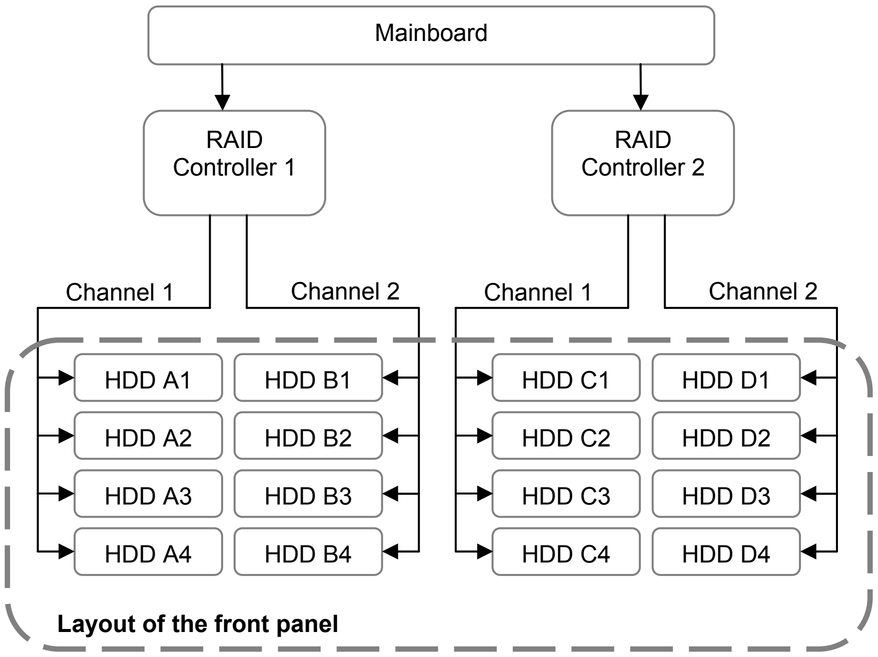


Figure 3-4: RAID system in the HP ProLiant DL380 G6

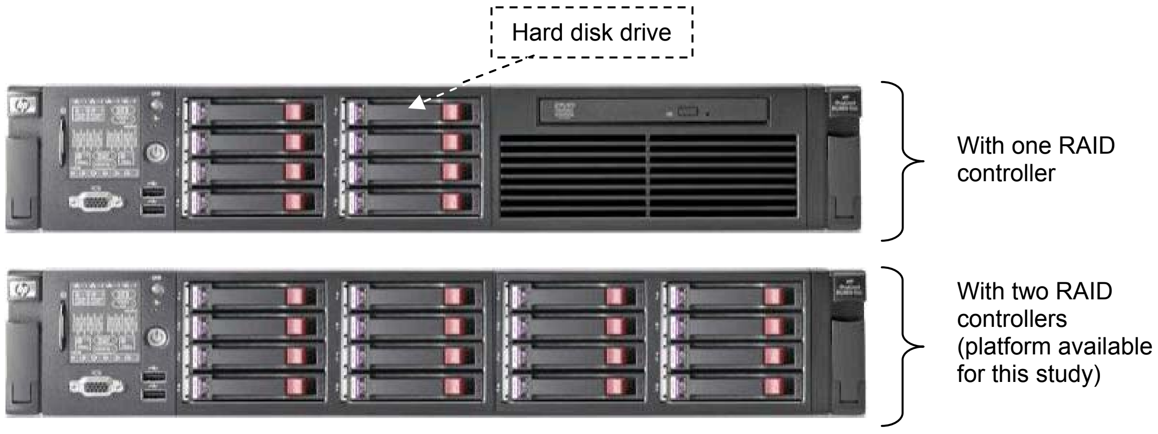

Hard disk drive
With one RAID controller
With two RAID controllers (platform available for this study)

Figure 3-5: Different layouts of the front panel of the HP ProLiant DL380 G6

With this platform it's possible to execute 16 threads at the same time (in parallel). For now, following vocabulary will be used:

- this platform has two *processors* (Intel Xeon E5520);
- each processor has 4 *cores*;
- each core has 2 *execution units*.

As shown in the next test (chapter 3.5.1.1.1 Overhead on CPU-Bound applications), each processor really is able to execute 8 threads in parallel without any loss of speed, so that we can consider that each processor has 8 execution units. Each core uses the hyper-threading technology[1] to be able to run 2 threads in parallel. [C14] [C16] [VV19]

[1] Information about this technology can be found in *Mikrocontroller und Mikroprozessoren.* [C5]

3.5.1 VMware ESXi

We first tested VMware ESXi 4.0.

3.5.1.1 CPU Utilization

3.5.1.1.1 Overhead on CPU-Bound applications

This test aims to measure the *approximate* overhead when running a *CPU-bound* [C8] program (little or no memory and I/O access) over the hypervisor. We compared the execution time of a test program on following platforms:

- A platform with simply a standard installation of Fedora Core 11 32-bits (i386) [C17], without virtualization. This Linux distribution is the most recent one delivered by Comsoft to its customers.
- A platform with VMware ESXi 4.0 Update 1 [VV11]. Here are the relevant parameters for this test:
  - 10 identical virtual machines are created, with following parameters:
    - Guest OS: "Other 2.6x Linux-System (32-Bit)"; [1]
    - Amount of virtual processors: 8 (maximum value), this means that the guest operating system will have the impression that it's working with 8 *execution units* and will be able to run 8 threads in parallel.
    - VMI Paravirtualization support: enabled.
    - CPU/MMU virtualization: automatic.
  - On each virtual machine a standard installation of Fedora Core 11 32-bits is performed.

The maximum amount of virtual processors should be 16, because the `htop` command (interactive process viewer) on the platform without virtualization confirms that a platform has 16 execution units. It should be possible to make a virtual machine use all of them. ESXi could be limited to 8 *virtual execution units* per virtual machine. This point will be clarified further.

Concerning VMI paravirtualization (Virtual Machine Interface), the document *Performance of VMware VMI* [VV12] shows that this option should bring better performances, and the Linux kernel of Fedora Core 11 32-bits supports this feature by default. You can see it by displaying the current kernel configuration after having installed Fedora on a machine:

```
$ vim "/boot/config-`uname -r`"
```

You should find the following line:

```
CONFIG_VMI=y
```

More information can be found in *Enabling VMI in a Linux kernel and in ESX* [VV16]. Because of a recent announcement from VMware of a phased retirement of the VMI functionality [VV17] [VV18], every test will also be performed without VMI paravirtualization support (it will be mentioned).

[1] Some terms may not be identical with the English version of VMware vSphere Client. We used the German version.

Structure of the test program: (the full implementation is available on the CD-ROM: `/eval/cpu/overhead/impl1/`)

```
create T threads executing
{
     // just the processor is needed:
     for (i = 0; i < M million; i++);
}
wait for the end of each thread;
display the execution time;
```

This test program is CPU-bound because:

- It's small enough to fit in the instruction cache (L1 instruction cache size = 32KB per core [C15]), thus few memory accesses are needed to fetch the instructions.
- It accesses such a small amount of data that they can fit in the data cache (L1 data cache size = 32KB per core [C15]).
- The only I/O access is the display of the result at the end.

Of course, this measurement is biased by following imperfections:

- the test process is not the only process launched on the system. However before launching the test as few processes as possible are present on the system and the `htop` command shows that on both systems, only one process is running[1] (actually `htop`), and only one execution unit is used (2% usage, actually because of `htop`). So we can reasonably consider that the effect of other processes on the test result will be negligible.
- The processor is regularly "disturbed" by interrupts coming from other components (I/O devices, timers...), but as we saw with `htop`, handling those events requires less than 1% of one execution unit. We can reasonably consider that this effect is negligible.
- The test program itself makes some system calls, in order to create threads for example, but we will set the parameters *T* and *M* (see the previous algorithm) so that this effect is negligible (less than 50 threads, in comparison to more than 30 seconds of execution).
- The Linux kernel's scheduler is regularly executed and may execute code triggering *VM Exits* on the system running ESXi (see appendix A Implementing full virtualization), but since it happens relatively rarely (for a process with normal priority, the *time-slice* is 100 ms, which is very long, especially for such a powerful machine [C9]), it is negligible. We could have set the maximum priority to this test program (time-slice = 800 ms) but, again, this is an approximate test.

No significant difference is expected between both configurations, since ESXi is not an emulator (see chapter 3.4.1 VMware Infrastructure). However, a difference might exist, because ESXi has to be executed regularly to schedule the different virtual machines and to update their timers [VV10]. The goal of this test is to check that this difference is negligible.

[1] htop calls them *running tasks* but it should actually call them *ready&running tasks*. A ready task is a task ready to be executed but waiting for the processor to do so (not waiting for any resource except for an execution unit). A running task is a task being run by a processor. (See [C7] for more details.)

Part 1

The test was run with M = 10 000, different numbers of threads (T) and 5 times for each value of T. Here are the results:

Number of threads	Execution time (without virtualization)	Execution time (over VMware ESXi)	Overhead
1	30.1 ± 0.5 s (σ_X = 0.55 s)	31.7 ± 0.5 s (σ_X = 0.45 s)	5.3 %
2	30.3 ± 0.5 s (σ_X = 0.84 s)	31.5 ± 0.5 s (σ_X = 0 s)	4.0 %
4	30.9 ± 0.5 s (σ_X = 0.55 s)	31.5 ± 0.5 s (σ_X = 0 s)	1.9 %
8	31.1 ± 0.5 s (σ_X = 0.55 s)	31.7 ± 0.5 s (σ_X = 0.45 s)	1.9 %
12	35.3 ± 0.5 s (σ_X = 0.45 s)	48.3 ± 0.5 s (σ_X = 0.84 s)	No comparison possible[1]
16	35.7 ± 0.5 s (σ_X = 0.45 s)	64.1 ± 0.5 s (σ_X = 1.1 s)	No comparison possible
20	47.7 ± 0.5 s (σ_X = 0.84 s)	79.1 ± 0.5 s (σ_X = 0.55 s)	No comparison possible
32	72.3 ± 0.5 s (σ_X = 2.0 s)	126.1 ± 0.5 s (σ_X = 1.1 s)	No comparison possible

Table 3-3: Overhead on CPU-Bound applications (1 VM)

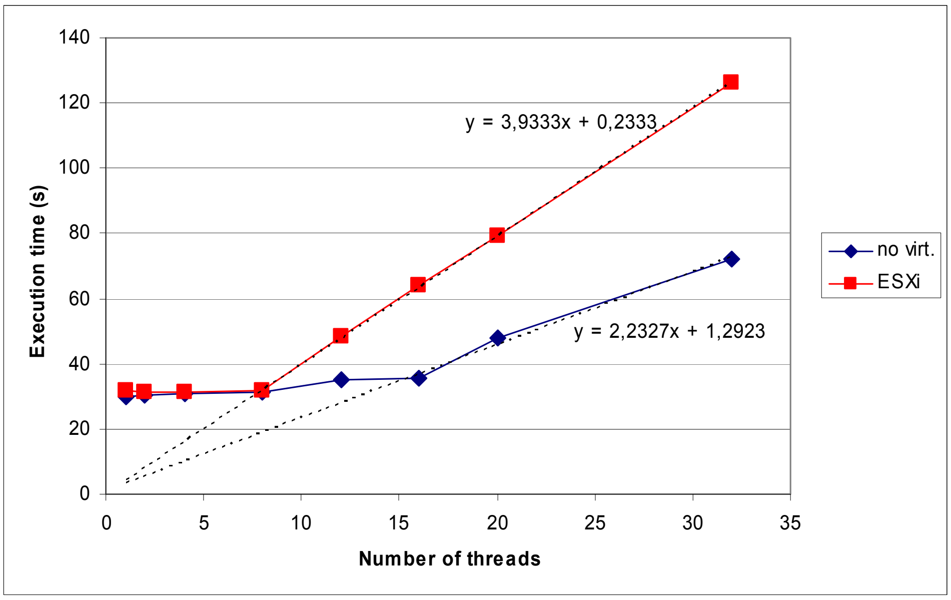


Figure 3-6: Overhead on CPU-Bound applications (1 VM)

Those results show that:

- On both systems, each thread is given about 100% of an execution unit to run, as long as there are more execution units than threads. Since all threads (of the test program) need the same time to complete. The execution of for example 1, 2 or 4 threads on a machine with 8 execution units takes exactly the same time. That's why the first part of each curve is horizontal.

[1] See below.

- When there are more threads than available execution units, the time needed to execute the whole program is then proportional to the number of threads, if the CPU time needed to schedule the amount of threads is negligible.
- In particular the system with ESXi shows a behaviour amazingly near to the theory explained in the two previous points. But the virtual machine only uses 8 execution units.
- **The maximum overhead caused by ESXi on a long execution of a CPU-bound program in one virtual machine is about 5 %** and appears when only one (Linux-) thread has to be scheduled, i.e. when only one execution unit is needed.
- The *standard deviation* doesn't seem to be increased because of ESXi. **ESXi doesn't add any significant indeterminism on a long execution of a CPU-bound program in one virtual machine.**

Note: the same result has also been obtained without enabling the "VMI paravirtualization support" for the virtual machine in ESXi.

Part 2

In the second part of this test, the test program is run simultaneously in 3 virtual machines, with T = 8 and M = 10 000. This gives us a total of 24 threads. Will ESXi then use the 16 available execution units?

- If yes, the expected execution time (without overhead in comparison to the previous result without virtualization) is: (see figure 3-6)

  2.2327 * 24 + 1.2923 = **54.9 s**

- If not (only 8 execution units used), the expected execution time (without overhead in comparison to the previous result using ESXi) is: (see figure 3-6)

  3.9333 * 24 + 0.2333 = **94.6 s**

The three instances of the program should be started at the same time, give or take 0.5 s, to keep a sufficient precision. To do so, we synchronize the machines with NTP (Network Time Protocol).

The test program has been modified. Here is its new structure: (the full implementation is available on the CD-ROM: `/eval/cpu/overhead/impl2/`)

```
wait until a given time;
create T threads executing
{
     // just the processor is needed:
     for (i = 0; i < M million; i++);
}
wait for the end of each thread;
display the execution time;
```

The test was repeated 5 times. The results are a mean execution time of 110.4 s and a standard deviation of 0.83 s. They show that:

- **ESXi still uses only 8 execution units** (otherwise the execution time would have been shorter).
- The mean **overhead** induced by ESXi on a long execution of CPU-bound programs in 3 virtual machines is **16.7 %** (in comparison to the result with only one virtual machine), which is **not negligible.**

- The *standard deviation* doesn't seem to be increased because of ESXi. **ESXi doesn't add any significant indeterminism on a long execution of CPU-bound programs in 3 virtual machines.**

Note: without enabling the "VMI paravirtualization support" for the virtual machine in ESXi, another result has been obtained:

- better mean execution time: 100.3 s (➔ 6.0 % overhead);
- worse standard deviation: 1.6 s;
- as shown in detail in the next part, a preference for one virtual machine (the third one) by the scheduler has been observed.

Part 3

In the third step, the same test with 10 virtual machines is performed. The expected execution time (without overhead in comparison to the execution time in only one virtual machine) is: (see figure 3-6)

3.9333 * (10 * 8) + 0.2333 = **314.9 s**

Here are the results:

		Execution time (± 0.5 s) / VM N°										Per test (s)	
		1	2	3	4	5	6	7	8	9	10	Mean	σ_X
Test N°	1	195.5	196.5	196.5	197.5	191.5	196.5	195.5	192.5	193.5	192.5	194.8	2.1
	2	194.5	196.5	195.5	195.5	190.5	194.5	194.5	191.5	192.5	191.5	193.7	2.0
	3	195.5	193.5	195.5	195.5	191.5	194.5	193.5	191.5	192.5	192.5	193.6	1.6
	4	194.5	193.5	195.5	196.5	192.5	194.5	195.5	192.5	191.5	192.5	193.9	1.6
	5	194.5	194.5	194.5	195.5	191.5	193.5	195.5	192.5	192.5	191.5	193.6	1.5
Per VM (s)	Mean	194.9	194.9	195.5	196.1	191.5	194.7	194.9	192.1	192.5	192.1	Result: 193.9 s (σ_X = 1.8 s) ➔ 7.8 % overhead	
	σ_X	0.55	1.52	0.71	0.89	0.71	1.1	0.90	0.55	0.71	0.55		

Table 3-4: Overhead on CPU-Bound applications (10 VMs)

Those results show that:

- **This time, ESXi uses the 16 execution units.** So the expected execution time (without overhead in comparison to the execution time without virtualization) with 16 execution units would have been: (see figure 3-6)

  2.2327 * (10 * 8) + 1.2923 = **179.9 s**

- The mean **overhead** induced by ESXi on a long execution of CPU-bound programs in 10 virtual machines is **7.8 %** (in comparison to the execution time without virtualization).

- **The *standard deviation* per virtual machine is still small** but the global one has become a little bigger. The mean execution time per virtual machine shows the reason: **the ESXi scheduler seems to prefer some virtual machines**, even if this difference remains small. For example, virtual machine 5 always gets more CPU time than number 4.

The problem discovered in this part of the test is that, on this platform, **it can't be predicted if ESXi will use 8 or 16 execution units**, whereas the VMware Compatibility Guide [VV15] shows that our platform ("HP ProLiant DL380 G6 – Intel Xeon 55xx Series") is supported by our version of ESXi ("ESXi 4.0 Installable U1").

We tried to change the "CPU Affinity" of the virtual machines (properties / resources / CPU advanced) to force some virtual machines to use the 8 first execution units ("0-7") and some others to use the 8 last ones ("8-15"). ESXi recognizes that there are 8 cores and accepts execution units from 0 to 15, but the number of execution units actually used stays unpredictable. We haven't found any document about this problem. Thus we would admit that **ESXi is unpredictable when used on 2 Intel Xeon E5520 processors when Hyperthreading is enabled.**

Part 4

To bypass this problem, the same tests were run:

- on the same server, but with hyperthreading disabled in the BIOS (2 processors without hyperthreading = 8 execution units);
- on another G6 server, exceptionally available at that time, having only one Intel Xeon E5520 (1 processor with hyperthreading = 8 execution units).

This part focuses on running the tests on the hardware platform with only one Intel Xeon E5520. A note indicates if similar results have been obtained on the other platform (2 processors without hyperthreading). However two Intel Xeon E5520 without hyperthreading may be more efficient than the other platform because:

- Without hyperthreading, each execution unit has its own L1 and L2 cache (it doesn't have to share it with another one), since there is one execution unit per core;
- With two processors, two MMU are available, since there is one MMU per processor.

Here is the result of the comparison between the mean execution time of a CPU-bound application on the machine without virtualization and the mean execution time over ESXi (one virtual machine):

Number of threads	Execution time (without virtualization)	Execution time (over VMware ESXi)	Overhead
1	30.9 ± 0.5 s ($σ_X$ = 0.55 s)	33.3 ± 0.5 s ($σ_X$ = 0.84 s)	7.8 %
2	30.9 ± 0.5 s ($σ_X$ = 0.55 s)	32.1 ± 0.5 s ($σ_X$ = 0.55 s)	3.9 %
4	30.9 ± 0.5 s ($σ_X$ = 0.55 s)	32.1 ± 0.5 s ($σ_X$ = 0.55 s)	3.9 %
8	35.7 ± 0.5 s ($σ_X$ = 0.45 s)	36.9 ± 0.5 s ($σ_X$ = 0.55 s)	3.4 %
12	53.7 ± 0.5 s ($σ_X$ = 0.45 s)	56.3 ± 0.5 s ($σ_X$ = 0.84 s)	4.9 %
16	70.7 ± 0.5 s ($σ_X$ = 0.45 s)	73.7 ± 0.5 s ($σ_X$ = 0.45 s)	4.2 %
20	89.9 ± 0.5 s ($σ_X$ = 0.89 s)	92.9 ± 0.5 s ($σ_X$ = 0.55 s)	3.3 %
32	142.1 ± 0.5 s ($σ_X$ = 1.5 s)	147.3 ± 0.5 s (σX = 0.84 s)	3.7 %

Table 3-5: Overhead on CPU-Bound applications (1 VM)

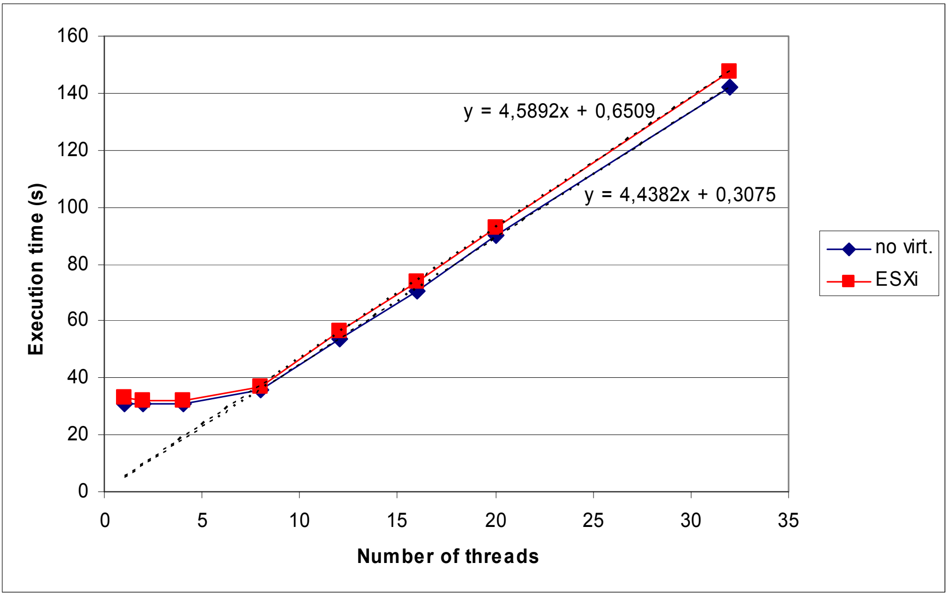


Figure 3-7: Overhead on CPU-Bound applications (1 VM)

This time, ESXi uses as many execution units as available (8 units), as Linux does. Conclusions stay about the same as in the first part of this test, which means:

- On both systems, each thread is given about 100% of an execution unit to run, as long as there are more execution units than threads. Since all threads (of the test program) need the same time to complete. The execution of for example 1, 2 or 4 threads on a machine with 8 execution units takes exactly the same time. That's why the first part of each curve is horizontal.
- When there are more threads than available execution units, the time needed to execute the whole program is then proportional to the number of threads, if the CPU time needed to schedule the amount of threads is negligible.

- **The maximum overhead caused by ESXi on a long execution of a CPU-bound program in one virtual machine is about 8 %** and appears when only one (Linux-) thread has to be scheduled, i.e. when only one execution unit is needed.
- The *standard deviation* doesn't seem to be increased because of ESXi. **ESXi doesn't add any significant indeterminism on a long execution of a CPU-bound program in one virtual machine.**

Note: the same result has also been obtained without enabling the "VMI paravirtualization support" for the virtual machine in ESXi.

Note: similar results have also been obtained with 2 processors without hyperthreading, but slightly better:

- no significant overhead;
- shorter execution time, for example about 122 s with 32 threads instead of 147 s.

Then, like before, the same test is run simultaneously on 3 virtual machines (➔ total of 24 threads) with T = 8 and M = 10 000. The expected execution time (without overhead in comparison to the execution time in one virtual machine) is: (see figure 3-7)

4.5892 * 24 + 0.6509 = **110.8 s**

The results are an execution time of 109.5 s (1.2 % improvement) without deviation. It was justified to perform every test again on a machine with only one Intel Xeon processor, because the results of the previous parts showed an overhead of about 17% when running 3 virtual machines, in comparison with the execution time when running only one virtual machine. Those new results actually show a **better CPU utilization when running 3 virtual machines than when running only one** (negative overhead).

The execution time is actually better than with two Intel Xeon E5520 processors with hyperthreading enabled. This confirms that **for now VMware ESXi doesn't correctly support two Intel Xeon E5520 processors with hyperthreading on the same platform.**

Those results also **show a high predictability** (no deviation was observed). So here again, the test leads to the conclusion that **ESXi doesn't add any significant indeterminism on a long execution of CPU-bound programs in 3 virtual machines.**

Note: the same result has also been obtained without enabling the "VMI paravirtualization support" for the virtual machine in ESXi.

Note: similar results have also been obtained with 2 processors without hyperthreading, but slightly better: The execution time is about 91 s, instead of 109.5 s.

Finally the same test with 10 virtual machines is performed. The expected execution time (without overhead in comparison to the execution time in only one virtual machine) is: (see figure 3-7)

4.5892 * (10 * 8) + 0.6509 = **367.8 s**

The results are a mean execution time of 368.0 s, a standard deviation of 1.7 s, and no observed preference for some virtual machines. They show that:

- **No overhead** has been introduced for the scheduling of 10 virtual machines performing a long execution of CPU-bound programs (in comparison to the execution time in only one virtual machine).
- The preference for some virtual machines may have been a consequence of the bad handling of our two processors with hyperthreading.

- The global standard deviation still remains very small (1.7 seconds in comparison with a mean execution time of 368 seconds). We can conclude that **ESXi doesn't add any significant indeterminism on a long execution of CPU-bound programs.**

Note: the same result has also been obtained without enabling the "VMI paravirtualization support" for the virtual machine in ESXi.

Note: similar results have also been obtained with 2 processors without hyperthreading, but slightly better: The execution time is about 310 s, instead of 368.0 s.

The results of the four parts of this test are **satisfactory**, except the unpredictable handling of two Intel Xeon E5520 processors with hyperthreading.

3.5.1.1.2 Overhead on system calls

This test aims to measure the *approximate* overhead when performing system calls. We compared the execution time of a test program on following systems:

- A system without virtualization (the one used for the previous test, with Fedora Core 11);
- A system with VMware ESXi (the one used for the previous test).

The hardware platform with only one Intel Xeon Processor is still used. Again, a note indicates the result with two processors without hyperthreading.

As described in appendix A Implementing full virtualization, system calls can induce overhead while making the guest enter the kernel mode, which has to be handled by the hypervisor. The document *Performance of VMware VMI* [VV12] describes a microbenchmark called "getppid" stressing the *syscall entry and exit path*, thus being able to measure the system call overhead.

This test program is similar to the one used for the previous test, but on each iteration of the loop, it issues a *getppid* system call (returns the process ID of the parent of the calling process). Structure: (the full implementation is available on the CD-ROM: `/eval/cpu/overhead/sysc_mmu/`)

```
create T threads executing
{
    for (i = 0; i < K thousand; i++)
        getppid();
}
wait for the end of each thread;
display the execution time;
```

Like for the previous test, we assume that the way the results will be biased by imperfections already described is negligible.

The test was run with K = 100 000, different number of threads (T) and 5 times for each value of T. Figure 3-8 shows the results. The measured standard deviation was the same on both platforms (max. 0.84 s).

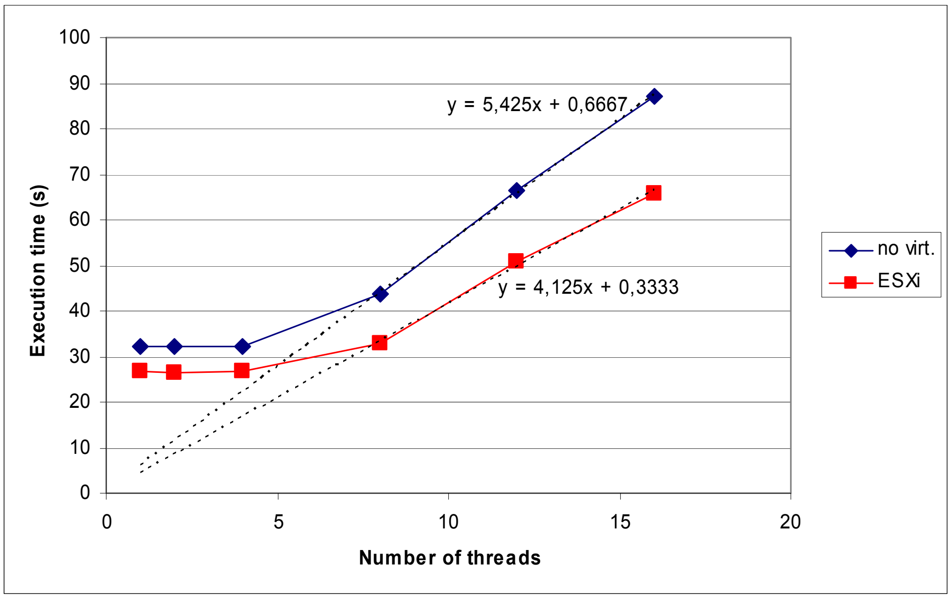


Figure 3-8: Overhead on system calls

Those results show that:

- VMware ESXi brings an **improvement of about 20 %** on this microbenchmark running as many system calls as possible. The most plausible explanation would be that VMI paravirtualization is the cause of this improvement (as explained in chapter 2.1.2.2 Virtualization Techniques, paravirtualization can lead to a performance better than native). But following tests were made on the configuration of the virtual machine in ESXi:
  - The parameter "support VMI Paravirtualization" can be set to *disabled* or *enabled*, it has actually no influence on the execution time. The assumption was wrong.
  - Setting the parameter "CPU/MMU-Virtualization" to "Automatic", "Intel VT-x/AMD-V for the virtualization of the instruction set and software for the virtualization of the MMU" or "Intel VT-x/AMD-V for the virtualization of the instruction set and Intel EPT/AMD RVI for the virtualization of the MMU" has no influence on the execution time either.
  - Setting this parameter to "software for the virtualization of the instruction set and the MMU" leads to a longer execution time (**about 120 % overhead**)

  All this just proves that the improvement has nothing to do with *instruction set virtualization* and *VMI paravirtualization*. But it doesn't prove that Intel EPT is responsible for this improvement, just that without it, things get worse. We can't explain why virtualization leads to an improvement of system call performance in this case.

- **ESXi doesn't add any significant indeterminism on a long execution of this microbenchmark.**

Performing the same test on *several* virtual machines is not relevant here, because:

- This microbenchmark is CPU bound: we can assume that the implementation of the Linux *getppid* system call doesn't heavily access the RAM. After this system call has been executed once, its result or the data needed to perform it should remain in *cache memory*.

- We now already know the behaviour of ESXi when running long CPU-bound applications.

The results of this test are more than **satisfactory**, because of this unexpected improvement.

Note: with 2 processors without hyperthreading, this improvement hasn't been observed. An overhead (max. 9 %) has been observed instead, which is still acceptable, since the predictability stayed very high.

3.5.1.1.3 MMU Performance

This test aims to measure the *approximate* overhead when heavily using the MMU[1]. The Intel Xeon E5520 Processor has the Intel EPT functionality (hardware support for MMU virtualization, contained in the Intel VT-x technology. [C14] We compared the execution time of a test program on following systems:

- A system without virtualization (the one used for the previous test, with Fedora Core 11);
- A system with VMware ESXi (the one used for the previous test).

The hardware platform with only one Intel Xeon Processor is still used. Again, a note indicates the result with two processors without hyperthreading.

The Linux *fork* instruction makes a copy of the calling process that differs from it only in its identifier and allocated resources (see the *fork manual page*). Its implementation uses the *copy on write* technique (both processes will share the same memory pages until one of both is modified) [C8]. Thus a simple call to *fork* doesn't need a big access to the memory; it just uses the MMU, because one more address space is created. The document *Performance of VMware VMI* [VV12] describes a microbenchmark called "nforkwait" stressing the MMU. It makes it possible to measure how good the virtualization of the MMU achieved in ESXi is.

This test program is similar to the one used for the previous test, but on each iteration of the loop, it creates a new process (with a *fork* system call) and waits for its termination (with a *waitpid* system call). It's also an implementation of the "fw" (fork-wait) test program used by VMware to evaluate the performance of Intel EPT Hardware Assist on an Intel Xeon. [VV13]

Structure of the test program: (the full implementation is available on the CD-ROM: `/eval/cpu/overhead/sysc_mmu/`)

```
create T threads executing
{
    for (i = 0; i < K thousand; i++)
    {
        fork();
        code of the child process
        {
            exit(0); // terminate
        }
```

[1] Memory Management Unit: hardware component (integrated into every Intel x86 processor since Intel 80386 [C3]) responsible for translating virtual addresses (one virtual address space per process, the word "virtual" has nothing to do with virtualization here) into physical addresses (one physical address space per machine) [C5]. With virtualization, one MMU per virtual machine has to be emulated if the hardware platform doesn't offer any support (e.g. Intel EPT [VV13] or AMD RVI [VV14]).

```
          code of the parent process
          {
               wait for the termination of the child;
          }
     }
}
wait for the end of each thread;
display the execution time;
```

Like for the previous test, we assume that the way the results will be biased by imperfections already described is negligible.

In this test we don't expect the execution time to scale with the number of threads like for the previous tests, since there is only one MMU per microprocessor (not one per execution unit).

The test was run with K = 10, different numbers of threads (T) and 5 times for each value of T. Here are the results:

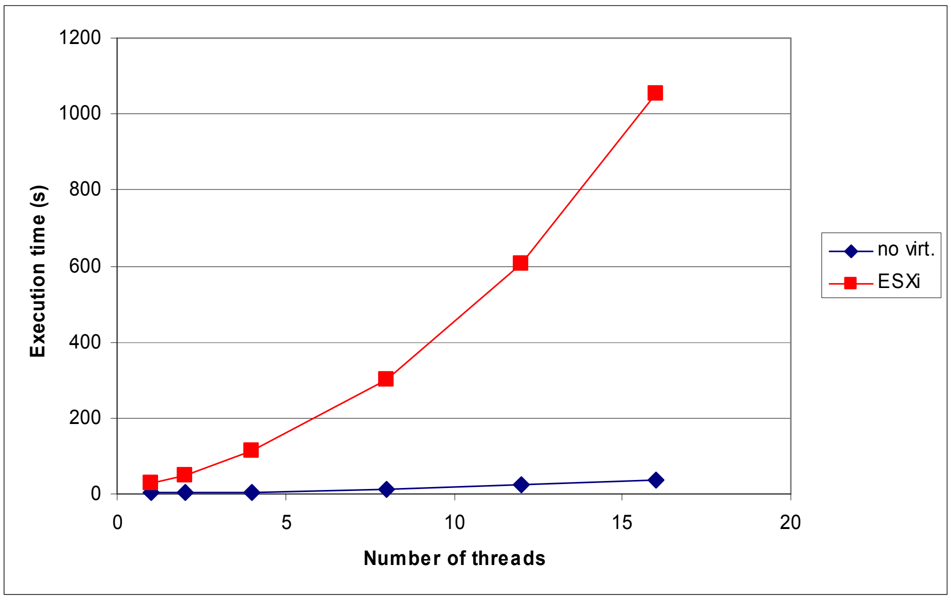


Figure 3-9: Overhead on MMU usage

The test should have been run with a bigger K, so that the execution time is greater than 30 seconds on both platforms and the results are more precise (as explained in chapter 3.5.1.1.1 Overhead on CPU-Bound applications), but the execution time would then have been huge on the platform using ESXi.

Those results show a **huge overhead**. Contrary to what one might expect, the "CPU/MMU-Virtualization" parameter (allowing to choose between a software-assisted MMU virtualization and a hardware-assisted one) has no influence on the result.

Finally VMI paravirtualization support has been disabled and much better results have been obtained: (this time the test has been run with K = 100)

Number of threads	Execution time (without virtualization)	Execution time (over VMware ESXi)	Overhead
1	29.9 ± 0.5 s (σ_X = 0.55 s)	51.1 ± 0.5 s (σ_X = 0.55 s)	71 %
2	34.9 ± 0.5 s (σ_X = 0.55 s)	63.3 ± 0.5 s (σ_X = 0.45 s)	81 %
4	54.1 ± 0.5 s (σ_X = 0.55 s)	104.5 ± 0.5 s (σ_X = 0.71 s)	93 %
8	134.5 ± 0.5 s (σ_X = 1.2 s)	246.5 ± 0.5 s (σ_X = 0.71 s)	83 %
12	239.3 ± 0.5 s (σ_X = 1.9 s)	424.7 ± 0.5 s (σ_X = 1.8 s)	77 %
16	367.5 ± 0.5 s (σ_X = 2.2 s)	642.1 ± 0.5 s (σ_X = 4.0 s)	75 %

Table 3-6: Overhead on MMU usage (without VMI paravirtualization)

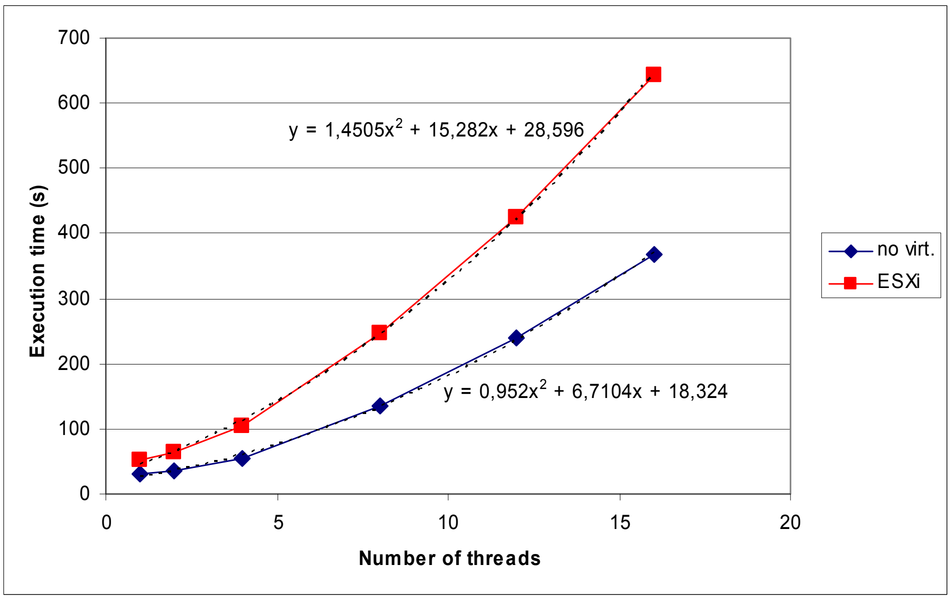


Figure 3-10: Overhead on MMU usage (without VMI paravirtualization)

Those results show that:

- As expected, the execution time doesn't scale well with the number of threads (execution time is not proportional to the number of threads), because there is only one MMU per processor, which acts as a bottleneck. The curves seem to be polynomial.
- The **overhead** is now reasonable but still **not negligible**. It's important to note that the overhead seems to **have a maximum**, which is **about 93 %**.
- **ESXi doesn't add any significant indeterminism on a long execution of this microbenchmark.** (A standard deviation of 4 seconds is actually still small in comparison to an execution time of 642 seconds.)

The important conclusions of this test are:

- **VMI paravirtualization shouldn't be used on this platform;**
- Even if ESXi generates a significant overhead on MMU virtualization, this overhead has a maximum and no significant indeterminism is created. Therefore ESXi still seems to be a good candidate for this project.

Note: with 2 processors without hyperthreading, two MMUs are available, reducing the bottleneck. This is why the maximum overhead observed is only **38 %**.

3.5.1.1.4 CPU time distribution

As shown in chapter 3.5.1.1.1 Overhead on CPU-Bound applications, ESXi distributes CPU time equally. This test aims to answer the following question: does "equally" means "equally among virtual machines"?

During a certain time frame (relatively short for a human being and relatively long for a microprocessor, usually between 100 milliseconds and a few seconds) each virtual machine needs to execute a certain number of instructions[1]. Depending on how many instructions the CPU can execute in this time frame, the need of a virtual machine corresponds to a certain *amount of CPU time*, or *fraction of the CPU*. If the total CPU need (sum of the needs of all the virtual machines) is greater than 100 % (CPU overloaded), the hypervisor has to decide how to distribute the CPU time (fairly). Classical strategies for a fair CPU time distribution are:

- *Distribute CPU time equally among virtual machines* means that when the CPU is overloaded, every virtual machine gets the same amount of CPU time, except the ones which don't need that much, for example:

	CPU time request (need)				CPU time allocation			
	VM 1	VM 2	VM 3	VM 4	VM 1	VM 2	VM 3	VM 4
Example 1	100 %	100 %	100 %	100 %	25 %	25 %	25 %	25 %
Example 2	50 %	50 %	50 %	50 %	25 %	25 %	25 %	25 %
Example 3	50 %	60 %	70 %	80 %	25 %	25 %	25 %	25 %
Example 4	10 %	30 %	40 %	60 %	10 %	30 %	30 %	30 %

Table 3-7: Equal CPU time distribution among virtual machines

  This scheduling strategy is a way similar to the *fair-share scheduling* described in the book *Operating Systems* [C7].

- *Distribute CPU time equally among threads* means that when the CPU is overloaded, the amount of CPU time given to one virtual machine is proportional to its number of threads. For example on this hardware platform there are 8 execution units, and all created virtual machines are based on 8 virtual CPUs, so they also "see" 8 execution units. Normally if one virtual machine has 16 ready threads (waiting for one execution unit), the scheduler of its guest operating system will allow only 8 of them to run at the same time, thus this virtual machine is only requesting for 100 % of the CPU (8 execution units), not 200 %. But we can imagine that, with *paravirtualization* (see chapter 2.1.2.2 Virtualization Techniques), the guest operating system indicates to the hypervisor its number of ready threads, i.e. its *real* need. The hypervisor could then give CPU time to virtual machines proportionally to their number of ready threads, so that each thread gets the same CPU time, for example:

[1] We consider that all instructions have the same duration. It simplifies the problem and leads to the same result.

	Number of ready threads				CPU time allocation			
	VM 1	VM 2	VM 3	VM 4	VM 1	VM 2	VM 3	VM 4
Example 1	8	8	8	8	25 %	25 %	25 %	25 %
Example 2	16	16	16	16	25 %	25 %	25 %	25 %
Example 3	8	8	16	32	12.5 %	12.5 %	25 %	50 %

Table 3-8: Equal CPU time distribution among threads

- *Distributing CPU time among virtual machines proportionally to their needs*:

	CPU time request				CPU time allocation			
	VM 1	VM 2	VM 3	VM 4	VM 1	VM 2	VM 3	VM 4
Example 1	100 %	100 %	100 %	100 %	25 %	25 %	25 %	25 %
Example 2	50 %	50 %	50 %	50 %	25 %	25 %	25 %	25 %
Example 3	50 %	60 %	70 %	80 %	19.2 %	23.1 %	26.9 %	30.8 %
Example 4	10 %	30 %	40 %	60 %	7.1 %	21.4 %	28.6 %	42.9 %

Table 3-9: Proportional CPU time distribution among virtual machines

Those three strategies can be considered as *fair*, but **in our project an equal CPU time distribution among virtual machines would be preferable**. As you can see in the previous examples, **this strategy is the only one guaranteeing** that a virtual machine gets **a reasonable minimal amount of CPU time** (25 % in this example) if it needs it.

To find out which strategy ESXi implements, several tests have to be run.

Part 1

Two virtual machines are created (A and B). The hardware platform with only one Intel Xeon processor is used again (8 execution units). Virtual machine A will run the test program presented in the first part of chapter 3.5.1.1.1 Overhead on CPU-Bound applications with M = 10 000 (ten thousand iterations per thread) and $T = T_A = 8$ (8 threads). Virtual machine B will run the same program with M = 10 000 and $T = T_B = 16$.

Let's define following variables and constants:

- `I`: number of CPU instructions in one iteration of the loop.
- $y_X(t)$: number of instructions per second the virtual machine X can execute at time t.
- Y: number of instructions per second the processor is able to execute.
- u_X: time needed to execute the program of the virtual machine X.

Expected execution times for each strategy:

- <u>ESXi distributes CPU time equally among threads: (assumption)</u>

$$y_A(t)=\frac{T_A}{T_A+T_B}\times Y \quad and \quad y_B(t)=\frac{T_B}{T_A+T_B}\times Y$$
$$with\, t\in[0,u_F],\ F=virtual\ machine\ whose\ program\ ends\ first$$

Let's determine which virtual machine finishes first:

$$A\ would\ finish\ at\ t=\frac{\#instructions_A}{y_A(0)}=\frac{T_A\times I\times M}{y_A(0)}=\frac{(T_A+T_B)\times I\times M}{Y}$$
$$B\ would\ finish\ at\ t=\frac{\#instructions_B}{y_B(0)}=\frac{T_B\times I\times M}{y_B(0)}=\frac{(T_A+T_B)\times I\times M}{Y}$$

$$\Rightarrow\ A\ and\ B\ end\ at\ the\ same\ time\ \Rightarrow \begin{Bmatrix} F=A=B \\ u_A=u_B \end{Bmatrix}$$

So if both programs finish at the same time, this means that ESXi distributes CPU time equally among threads.

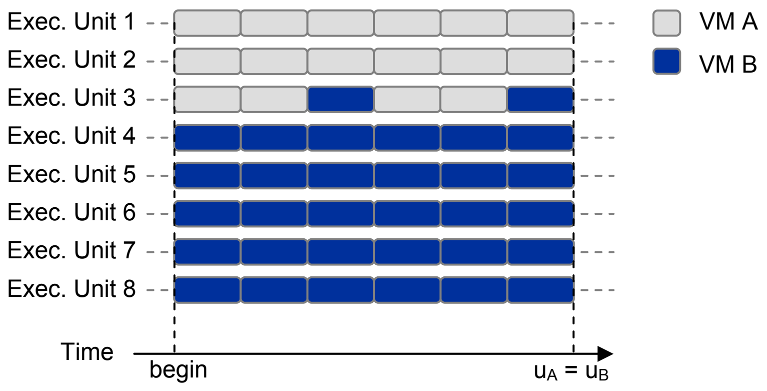


Figure 3-11: Equal CPU time distribution among threads

- <u>ESXi distributes CPU time equally among virtual machines: (assumption)</u>

$$y_A(t)=y_B(t)=\frac{Y}{\#VM}=\frac{Y}{2}$$
$$with\, t\in[0,u_F],\ F=virtual\ machine\ whose\ program\ ends\ first$$

Let's determine which virtual machine finishes first:

$$A \text{ would finish at } t = \frac{\#instructions_A}{y_A(0)} = \frac{T_A \times I \times M}{y_A(0)} = \frac{2 \times T_A \times I \times M}{Y}$$

$$B \text{ would finish at } t = \frac{\#instructions_B}{y_B(0)} = \frac{T_B \times I \times M}{y_B(0)} = \frac{2 \times T_B \times I \times M}{Y}$$

$$now\ T_A < T_B \ \Rightarrow\ F = A \ \Rightarrow\ u_A = \frac{2 \times T_A \times I \times M}{Y}$$

Let's determine u_B:

- *during* $t \in [0, u_A]$, *B has executed*
  $i_{done} = y_B(0) \times u_A = T_A \times I \times M$ *instructions*

  so there are still
  $i_{tbd} = T_B \times I \times M - i_{done} = (T_B - T_A) \times I \times M$ *instructions to be done*
- *on* $t \in]u_A, u_B]$, *B gets the whole CPU:* $y_B(t) = Y$

  remaining time to execute i_{tbd}*:*
  $$\Delta u_{tbd} = \frac{i_{tbd}}{y_B(u_B)} = \frac{(T_B - T_A) \times I \times M}{Y}$$
- $u_B = u_A + \Delta u_{tbd} = \dfrac{(T_A + T_B) \times I \times M}{Y}$

  $$\Rightarrow\ \frac{u_B}{u_A} = \frac{T_A + T_B}{2 \times T_A} = \frac{8+16}{2 \times 8} = 1.5$$

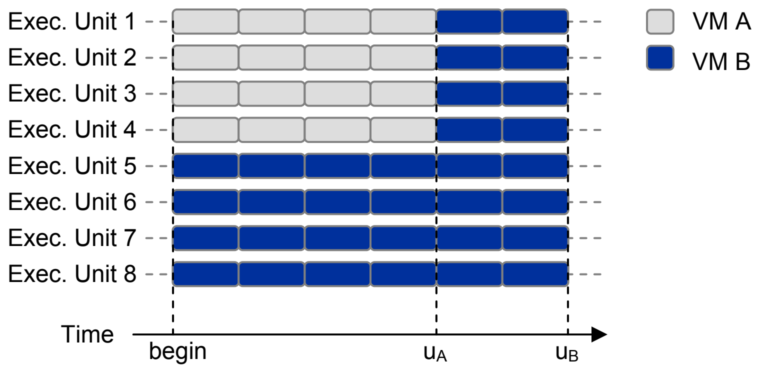


Figure 3-12: Equal CPU time distribution among virtual machines

- ESXi distributes CPU time proportionally among virtual machines: (assumption)

  Since both virtual machines have at least 8 threads, they both request 100 % of the CPU time. Thus both of them will get the same amount of CPU time (50 %):

  $$y_A(t)=y_B(t)=\frac{Y}{2}$$
  $$with\, t\in[0,u_F],\quad F=virtual\ machine\ whose\ program\ ends\ first$$

  Like for the previous strategy, A will finish first and B will need 50 % more time. So if the program on virtual machine B needs 50 % more time that the one on virtual machine A, this means that ESXi distributes CPU time either equally or proportionally among virtual machines.

This experiment showed that u_B = 1.5 x u_A: **ESXi distributes CPU time either equally or proportionally (to what they need) among virtual machines.**

The next experiment should decide between equal and proportional distribution.

Part 2

This test is the same as before, but with T_A = 4 and T_B = 16.

Expected execution times for each strategy:

- ESXi distributes CPU time equally among virtual machines: (assumption)

  Like for the previous test, A finishes first and:

  $$\frac{u_B}{u_A}=\frac{T_A+T_B}{2\times T_A}=\frac{4+16}{2\times 4}=2.5$$

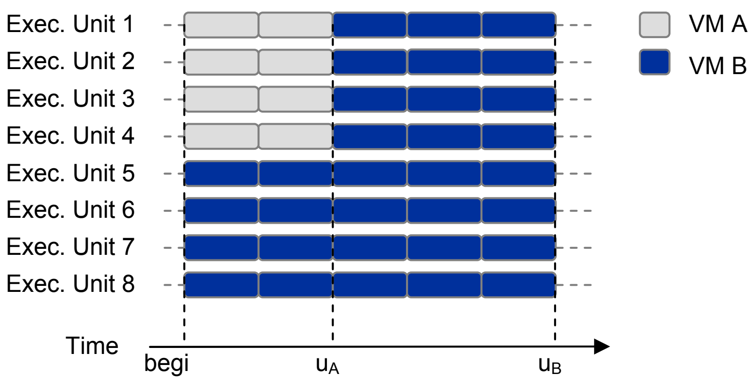


Figure 3-13: Equal CPU time distribution among virtual machines

- <u>ESXi distributes CPU time proportionally among virtual machines: (assumption)</u>
  - E: number of available execution units on the hardware platform. (= 8)
  - $e_A(t)$: number of execution units wanted by A at time t.

$$\text{with}\, t\in[0,u_F],\ F=\text{virtual machine whose program ends first}$$

$$e_A(t)=min(T_A,E)=min(4;8)=4$$

$$e_B(t)=min(T_B,E)=min(16;8)=8$$

$$y_A(t)=\frac{e_A(t)}{e_A(t)+e_B(t)}\times Y=\frac{Y}{3}$$

$$y_B(t)=\frac{e_B(t)}{e_A(t)+e_B(t)}\times Y=\frac{2Y}{3}$$

Let's determine which virtual machine finishes first:

$$A\ \text{would finish at}\ t=\frac{\#instructions_A}{y_A(0)}=\frac{T_A\times I\times M}{y_A(0)}=\frac{12\times I\times M}{Y}$$

$$B\ \text{would finish at}\ t=\frac{\#instructions_B}{y_B(0)}=\frac{T_B\times I\times M}{y_B(0)}=\frac{24\times I\times M}{Y}$$

$$\Rightarrow\ F=A\ \Rightarrow\ u_A=\frac{12\times I\times M}{Y}$$

Let's determine u_B:

- *during* $t\in[0,u_A]$, *B has executed* $i_{done}=y_B(0)\times u_A=8\times I\times M$ *instructions*

  so there are still $i_{tbd}=T_B\times I\times M-i_{done}=8\times I\times M$ *instructions to be done*

- *on* $t\in]u_A,u_B]$, *B gets the whole CPU:* $y_B(t)=Y$

  remaining time to execute i_{tbd}*:*

  $$\Delta u_{tbd}=\frac{i_{tbd}}{y_B(u_B)}=\frac{8\times I\times M}{Y}$$

- $u_B=u_A+\Delta u_{tbd}=\dfrac{20\times I\times M}{Y}$

  $$\Rightarrow\ \frac{u_B}{u_A}=\frac{20}{12}=\frac{5}{3}\simeq 1.67$$

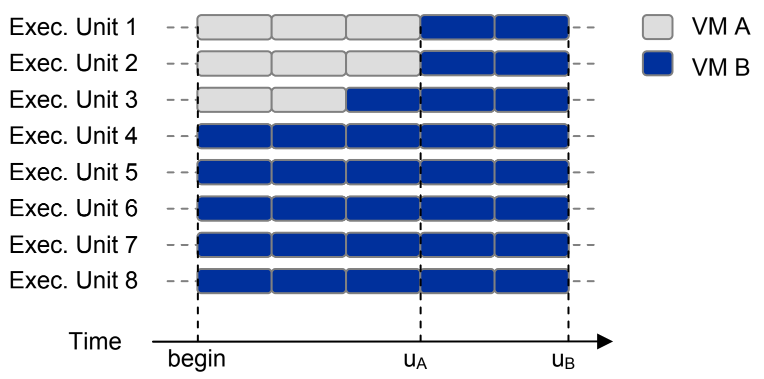


Figure 3-14: Proportional CPU time distribution among virtual machines

The same test has also been done with T_A = 4 and T_B = 8. Here are the results:

	Test 1 (T_A = 4, T_B = 16)	Test 2 (T_A = 4, T_B = 8)
Expected u_B/u_A if equal distribution	2.5	1.5
Expected u_B/u_A if prop. distribution	1.67	1
Observed u_B/u_A	**2.05**	**1.28**

Table 3-10: CPU time distribution

Note: similar results have also been obtained with 2 processors without hyperthreading.

There are at this point two possible explanations:

- VMware implemented a scheduling strategy leading to a hybrid CPU time distribution (between *equal distribution* and *proportional distribution*). However VMware claims to implement a scheduler distributing CPU time equally among virtual machines (if no *CPU reservation* or *limit* is set) in *The CPU Scheduler in VMware ESX 4.* [VV21] The author names it *Proportional-Share Based Algorithm* (not to mix up with *distributing CPU time among virtual machines proportionally to their number of threads*).
- The result is biased by some cache effects or scheduling overhead.

Part 3

This test aims to quantify the guaranteed amount of CPU time for a virtual machine.

With an equal distribution of the CPU time among virtual machines, the guaranteed amount of CPU time for a virtual machine only depends on the number of virtual machines, so that if there are 4 virtual machines, one of them has the guaranty to get 25 % of the CPU if it needs it (i.e. two of the eight execution units):

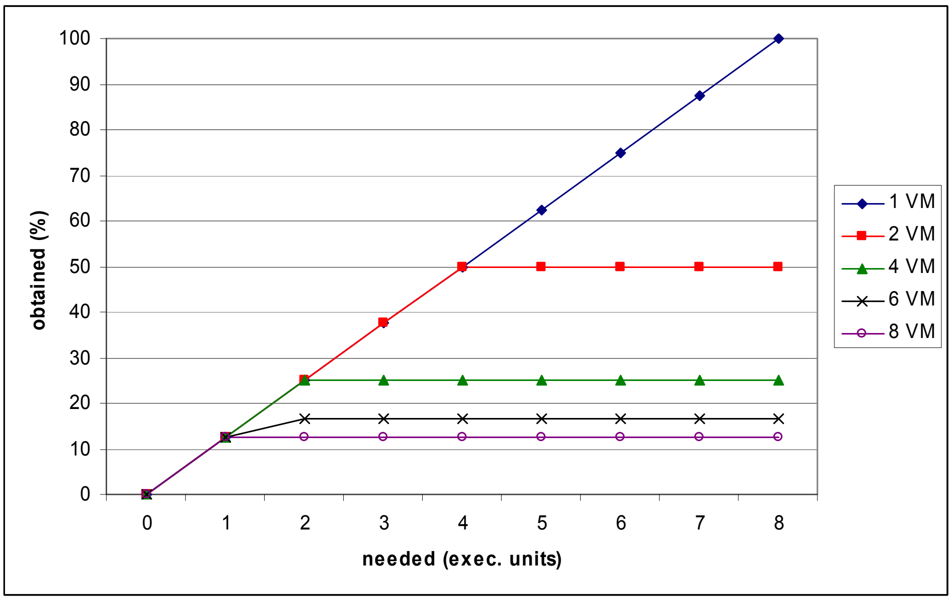


Figure 3-15: Equal CPU time distribution among virtual machines

A proportional distribution of the CPU time among virtual machines delivers less guaranty: if there are 4 virtual machines, three of them wanting all execution units, the last one wanting only 50 % of the CPU (4 execution units) will only get about 14 % of the CPU: (need / total need = 4 / (3 x 8 + 4) = 14.3 %)

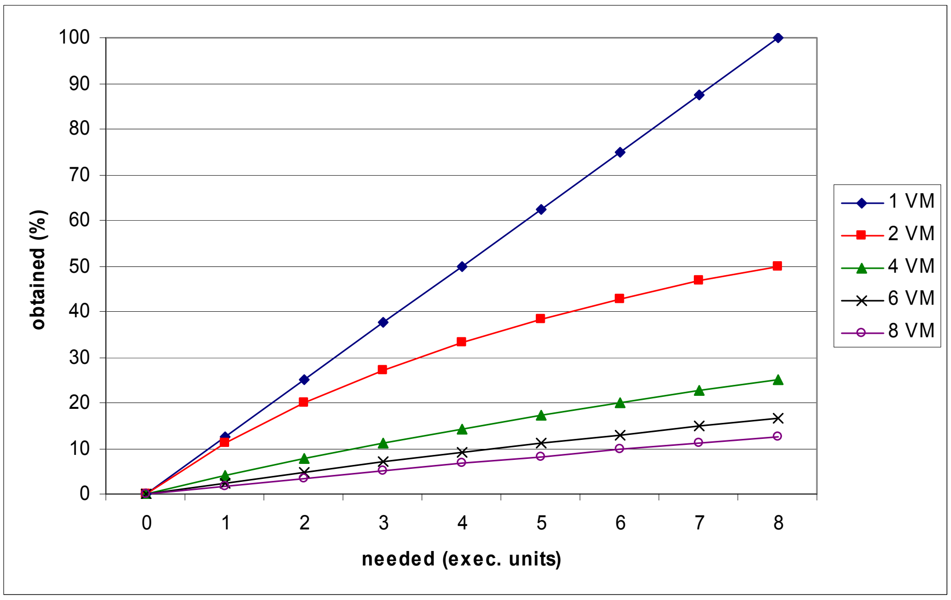


Figure 3-16: Proportional CPU time distribution among virtual machines

A proportional CPU time distribution gives advantage to virtual machines running a lot of threads, exactly like a normal Linux scheduler giving advantage to applications running a lot of threads. In this project, we are looking for a solution that is fair on the virtual machine level, not on the thread level, therefore an equal CPU time distribution is highly wanted.

To measure the real curves of the ESXi scheduler, we use the execution time without virtualization as reference. For example, if the program with 4 threads gets run in 20 seconds without virtualization (reference time), and if it gets run in 30 seconds with virtualization whereas a second virtual machine needs all execution units, then *a virtual machine wanting 4 execution units will obtain at least 33 % of the CPU time*:

(ref. time / exec. time) x percentage of CPU wanted = (20 / 30) x 50 % = 33.3 %.

Each test has been done five times. Here are the results:

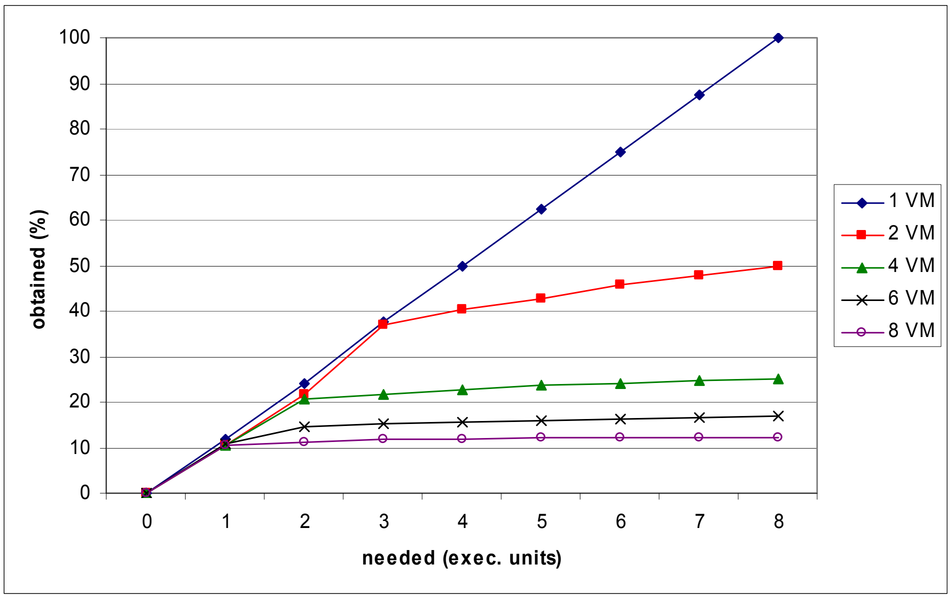


Figure 3-17: ESXi's CPU time distribution among virtual machines

Those results show that the CPU time distribution done by ESXi it a slightly smoothed version of the *equal distribution*, nevertheless **delivering following simple guarantees**:

- with 2 virtual machines, each virtual machine gets at least 37 % of the CPU if it needs it;
- with 4 virtual machines, each virtual machine gets at least 22 % of the CPU if it needs it;
- with 6 virtual machines, each virtual machine gets at least 14 % of the CPU if it needs it;
- with 8 virtual machines, each virtual machine gets at least 10.5 % of the CPU if it needs it.

At this point, **ESXi** is still **suited to this project**.

To go further, ESXi provides the possibility to define a ***CPU reservation*** for each virtual machine. With this feature, you can assign for example 70 % of the CPU time to one virtual machine. This feature is not immediately needed for this project, since we will give the same importance to each virtual machine (we just want to use a virtualization technology to eliminate the risk that one application makes another one starve), but appendix B CPU reservation on VMware ESXi examines this feature and shows that it **doesn't work when hyperthreading is enabled** on our platform.

3.5.1.1.5 Scheduling

The hardware platform with only one Intel Xeon Processor is still used. We assume that the platform with 2 processors delivers similar results.

The idea of this test is simple but its precise implementation is not. The following program has to be launched:

```
for (i = 0; i < 1 000 000 001; i++)
{
     // calculate the elapsed time between the current iteration and
     // the previous one:
     newTime = currentTime();
     diff = newTime – oldTime;
     oldTime = newTime;

     save(diff); // save the result somewhere
}
display statistics;
```

The main idea is: the elapsed time between two iterations should be constant, except when the process is *de-scheduled* (no execution unit is available for its execution). The goal is to evaluate the scheduling strategy[1] of ESXi. If this program is running in a virtual machine, since it always "wants" to run, the virtual machine also does. The scheduler of ESXi has then to decide to give (or not) an execution unit to this virtual machine. This test is supposed to answer the following question: **If a virtual machine wants an execution unit to run, how much time does it have to wait in the worst case?**

However following constraints make the exercise more complicated:

- On a classical operating system, this process is not the only running one. So if the process is de-scheduled, it doesn't mean that the virtual machine has been de-scheduled by the ESXi scheduler. The process could just have been de-scheduled by the scheduler of its operating system (*guest* operating system) to let another process run. Thus running our test in such environment wouldn't make any sense, we need to be sure that a de-scheduling of the test program *means* a de-scheduling of the virtual machine. Following solutions are suitable:
  - The program doesn't run on top of a (guest) operating system. The virtual machine just boots directly on this program. This solution needs a lot of effort to be implemented, because for example we need the program to be able to communicate with us to deliver the results, and we would have somehow to develop "drivers" for I/O.
  - The program runs on top of a *monotasking* operating system (e.g. MS-DOS). On such an operating system you have the guarantee that only one process is running at a time and that it can't be de-scheduled. However most monotasking operating systems are old now and their APIs (Application Programming Interfaces) are often very restrictive. This solution has been explored but **not retained** (see Appendix C Scheduling test on MS-DOS).
  - The program runs on top of a *hard real-time operating system*. The operating system must have a *priority-driven scheduler* (a process can only be de-scheduled if another one with a *higher* priority wants to run) and our test program must have the highest priority. This possibility seems to be the easiest one and **it's the one we have chosen**.
- The `save()` function can't save the results anywhere, because if it accesses a device:
  - Each iteration would last much longer than it should, thus distorting the results.

[1] Information on scheduling can be found in *Operating Systems* [C7] and *Echtzeitsysteme* [CR1].

  - The process would be de-scheduled on each iteration because it has to wait for the device (*interrupt-driven I/O* strategy [C7]).
  - The process would be waiting for the device most of the time, thus it wouldn't always "want" to run, as already mentioned.
  - Even accessing the RAM isn't adapted because it would probably be a bottleneck (there are several execution units) and we wouldn't be able to monitor the real performance. Moreover performing a RAM access is much longer than executing an arithmetic instruction [C7].

  The most suitable solution would be to store the results in a **small** structure (fitting in the L1 data cache). During the first iterations some *cache misses* will be encountered, until the structure entirely resides in the L1 data cache, thus slightly distorting the result at the beginning.[1]

- The `currentTime()` function has to deliver a value varying fast enough to be different between two successive iterations. Considering the frequency of actual microprocessors, the precision of the clock used by this function must be a few *nanoseconds*.

Several *hard-real time operating systems* suitable for this test are available for free:

- *RTLinux*: it's complicated to find a Linux kernel that can be patched, find the correct patch adapted to a specific kernel version and perform the installation. Then the program has to be compiled and loaded as *kernel module*, which is not clearly documented if you have some assembler files (`.s` files). Therefore we didn't choose it, but documents [CR2] [CR3] [CR4] [CR5] may be interesting if you want to try it.

- *MiniRTL*: small version of RTLinux designed to fit in a floppy disk. It's easier to install than RTLinux (the floppy disk image is already available) but it appeared to be unstable on our system (a lot of *segmentation faults* when starting and appearing at unpredictable moments). Therefore we didn't choose it, but the document [CR7] may be interesting if you want to try it.

- *Xenomai*: very active project [CR8] where the Linux kernel is patched to run in *user mode* as a normal application with the lowest priority for Xenomai's hard real-time kernel. [CR9] Any application can then be run on top of the Linux kernel (*not real-time*) or directly on top of Xenomai's kernel (*real-time*). **We chose this OS for this test.**

- *RTAI*: project based on the same principle as Xenomai. [CR10]

We followed exactly the instructions of the document [CR11] to install a *hard real-time operating system* based on a Linux Debian Lenny on each (virtual or real) machine. The document [CR12] shows some examples for programming in a *Xenomai* environment. The document [CR13] is the official Xenomai API Documentation.

The document *Timekeeping in VMware Virtual Machines* [VV10] describes in a very understandable way all the problems and solutions concerning time keeping on real and virtual machines. In particular it exposes all available hardware counters that could be used to implement the `currentTime()` function. The *TSC* (Time Stamp Counter) is adapted ("fine-grained and convenient"), and we are not concerned by the drawbacks highlighted in the document:

[1] Please note that every approximation made for this test will only make the results look *worse* than they really are. There is no way that the results look good if they aren't.

- we don't need to know exactly the frequency of the TSC, since we won't use it to keep time on a clock;
- there is no reason for the hardware components or for the operating system to change (slow down) the pace of the TSC during the test, since this test will keep the microprocessor busy the whole time (the pace usually gets slowed down to save energy when the computer has "nothing" to do);
- the TSC can be read in a virtual machine, since the implementation of the virtual TSC in ESXi "doesn't count cycles of code run on the virtual CPU but advances even when the virtual CPU is not running", which is what we need.

An example of implementation using the *PIT* (Programmable Interval Timer) instead of the TSC is available in Appendix C Scheduling test on MS-DOS.

The structure chosen to save the statistics is an array of 12 *integers*. On each iteration:

- If the difference of time between the current iteration and the previous one is smaller than 10 ticks of the TSC, the first box of the array is incremented;
- If the difference is between 10 and 100 ticks, the second box is incremented;
- ...
- If the difference is between 10^{k-1} and 10^{k}, the k^{th} box is incremented;
- ...
- If the difference is greater than 10^{11}, the 12th box is incremented.

The new algorithm looks like: (the full implementation is available on the CD-ROM: `/eval/cpu/xenomai/`)

```
for (i = 0; i < 1 000 000 001; i++)
{
     // calculate the elapsed time between the current iteration and
     // the previous one:
     newTime = rdtsc(); // Read TSC
     diff = newTime – oldTime;
     oldTime = newTime;

     if (diff > 10^11)
     {
          Result[11]++; // 12th box
     }
     else if (diff > 10^10)
     {
          Result[10]++;
     }
     // ...
     else if (diff > 10)
     {
          Result[1]++;
     }
     else
     {
          Result[0]++;
     }
}
display Result;
```

Because of the conditional structure, the duration of an iteration is not constant, depending on which branch is taken. Fortunately it should converge (stabilize) pretty fast, because:

- If one iteration *i* is long,
- *diff* will be significant on iteration *i+1*,
- so one of the first cases ("if, else if, else if, ...") will be taken at the end of iteration *i+1*,
- thus making the execution time of the iteration *i+1* shorter
- and *diff* smaller on iteration *i+2*.

In conclusion: the longer an iteration, the shorter the next one and vice versa.

Without virtualization

The test program was run five times on the platform running Xenomai without virtualization. The worst obtained results are:

[0, 10[	[10, 100[	[100, 1K[	[1K, 10K[	[10K, 100K[	[100K, 1M[	[1M, 10M[	[10M, 100M[	[100M, 1G[	[1G, 10G[	[10G, 100G[	[100G, ∞[
0	999 992 427	195	7 138	0	240	0	0	0	0	0	0

Table 3-11: Scheduling test on Xenomai without virtualization

This table means that the elapsed time between two consecutive iterations of the loop:

- has never been strictly lower than 10 ticks of the TSC timer;
- has been 999 992 427 times between 10 and 99 ticks;
- ...

We can see that the interval between two iterations has always been smaller than 1 million ticks and has been higher than 10 000 ticks only 240 times in 10 billion. Those 240 times have probably been caused by cache misses at the beginning or by disturbing interrupts coming from devices (e.g. from one NIC). Even if the operating system waits before calling the handlers corresponding to those interrupts (because a high priority task is running), it has to write somewhere in its own data structures that an interrupt occurred and it has to acknowledge the interrupt on the device.

When running this test several times, we can see that variation in the results is extremely low. This is because we use a real-time operating system. When running a real-time task on this system, all non real-time tasks are frozen. The system doesn't even react to the "Ctrl+C" combination: it isn't possible to stop the task anymore. That means that this operating system is exactly what we needed for this test: the only way to change the results is to add a virtualization layer (a hypervisor) under this operating system. We will be able to measure the influence of the hypervisor's scheduler.

On most processors the TSC's frequency (ticks per seconds) is the processors' frequency (for some exceptions it can be a multiple). A program can easily be written to approximate the TSC's frequency and confirm it (see the implementation on the CD-ROM: `/eval/cpu/tsc_freq/`). On this hardware platform:

- TSC's frequency = 2266 MHz;
- 1 tick = 0.441 ns.

So the interval between two instructions has always been smaller than 441 ms, and has been higher than 4.41 ms only 240 times in 10 billion. You might think that those 240 times in

10 billion are negligible, **but are they really?** In fact the approximate execution time of one of those 240 iterations (between 100 000 and 999 999 ticks) is **10 000 times higher** than the approximate execution time of one of the 999 992 427 shortest iterations (between 10 and 100 ticks).

That means that one execution time between 100 000 and 999 999 ticks has about **10 000 times** more *importance* than one between 10 and 100 ticks. Thus we have to define a coefficient called *importance* for each time range where:

$$importance(Range_{i+1}) = 10 \times importance(Range_i)$$

Time range	[0, 10[	[10, 100[	[100, 1K[	[1K, 10K[	[10K, 100K[	[100, 1M[	● ● ●
Importance	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	

● ● ●	[1M, 10M[	[10M, 100M[	[100M, 1G[	[1G, 10G[	[10G, 100G[	[100G, ∞[
	0.01	0.1	1	10	100	10^{3}

Table 3-12: Importance of each time range

We now define the ***approximate*** *influence* of the whole population of one range on the execution time as:

$$influence(Range_i) = size(Range_i) \times importance(Range_i)$$

The approximate influence can also be normalized, so that the sum of all normalized influences is 1:

$$\overline{influence_i} = \overline{influence}(Range_i) = \frac{influence(Range_i)}{\sum_k influence(Range_k)}$$

$$\Rightarrow \sum_k \overline{influence_k} = 1$$

Thus the result of this test can be represented by the following:

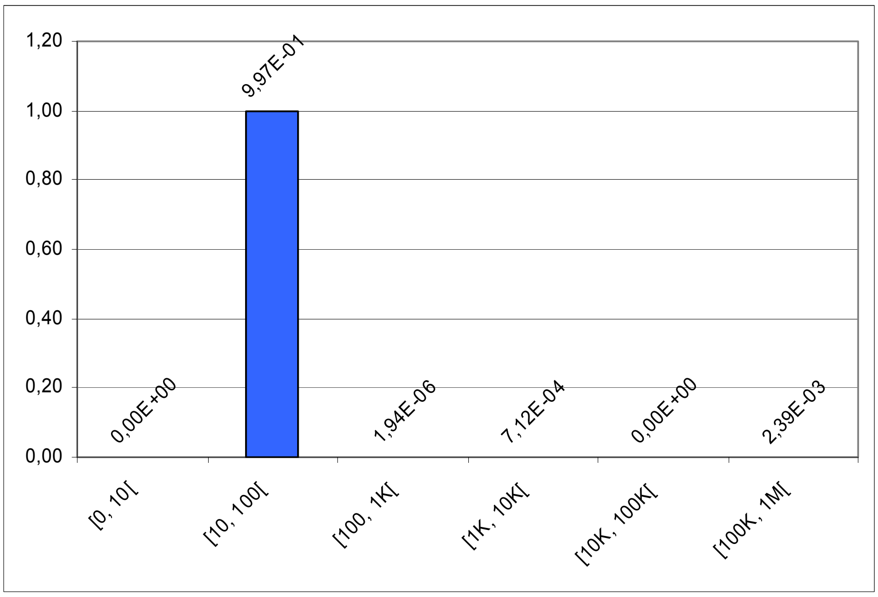


Figure 3-18: Distribution of the normalized approximate influence without virtualization

So we can conclude that **the 240 times** where the execution time between two iterations was greater than 100 000 ticks **are negligible**, since their approximate influence on the total execution time is only 0.239 %.

As explained in chapter 2.1.3.2 Soft Real Time, a *soft real-time system* doesn't need a perfect predictability (predictability = 100 %). Thus we can accept on the virtualization layer a predictability of only 95 %. In this test, aiming to measure the latency implied by the scheduler of ESXi, this means that we can ignore the longest measurements whose total influence is lower as 5 %. For example in this case, we can ignore measurements in the range [100, ∞[, because the total influence of this range is:

$1.94 \times 10^{-6} + 7.12 \times 10^{-4} + 2.39 \times 10^{-3} = 3 \times 10^{-3}$ = **0.3 %**

This is the biggest range that can be ignored, because the next one ([10, 100[ ) has an influence of 99.7 %. We can conclude that in this case (without virtualization and with Xenomai as operating system) **the execution time between two iterations** of the test program is in most cases **shorter than 100 ticks** (44.1 ns). The execution time can happen to be longer, but the impact on the whole execution time is about 0.3 %.

This upper bound for the execution time can be formally defined as:

$$bound_{95\,\%} = \min_{t \in 10^{\mathbb{N}}} \{ t \quad with \quad \overline{influence}([t, \infty[) \leqslant 5\,\% \}$$

and more generally:

$$p \in [0,1], \quad bound_p = \min_{t \in 10^{\mathbb{N}}} \{ t \quad with \quad \overline{influence}([t, \infty[) \leqslant (1-p) \}$$

Note: do not forget that this value is just an order of magnitude, since the whole calculation is based on powers of 10.

***bound**$_{95\%}$* is an interesting indicator as it does **not only guarantee that 95 % of the cases are under the bound**, but also that the **influence on the whole execution time** of the exceeding cases **is lower than 5 %**.

Appendix D Scheduling test on Fedora Core 11 shows the results of this test on Fedora Core 11 without virtualization, thus proving that Fedora couldn't have been used as a reference system for this test.

With one virtual machine

The test has been run five times on one virtual machine running Xenomai. The worst results are:

[0, 10[	[10, 100[	[100, 1K[	[1K, 10K[	[10K, 100K[	[100K, 1M[	[1M, 10M[	[10M, 100M[	[100M, 1G[	[1G, 10G[	[10G, 100G[	[100G, ∞[
0	997 518 966	2 463 507	9 635	7 364	528	0	0	0	0	0	0

Table 3-13: Scheduling test on one virtual machine

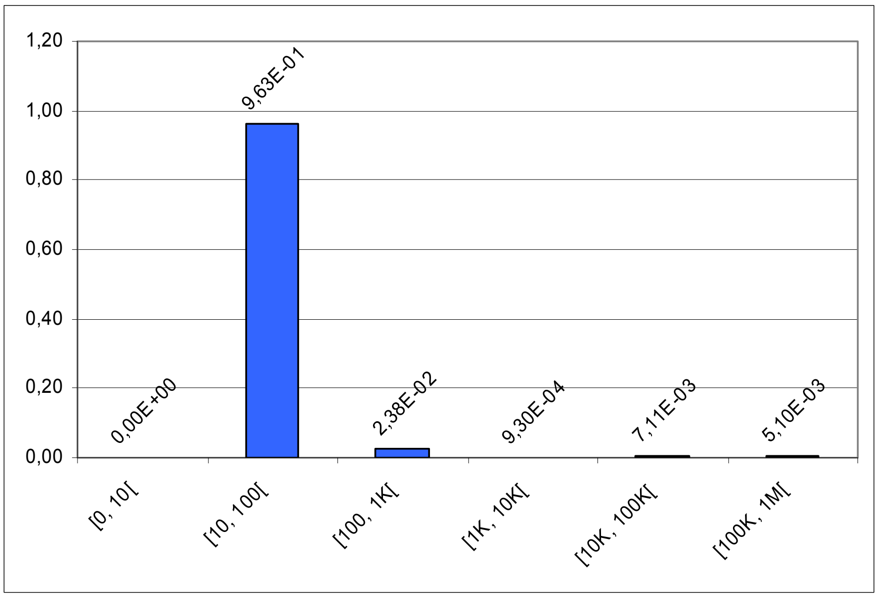


Figure 3-19: Distribution of the normalized approximate influence (1 VM)

Like without virtualization, bound$_{95\%}$ = 100 ticks = 44.1 ns. The order of magnitude is still the same. We can conclude that **ESXi doesn't generate any significant latency for the processor when running one virtual machine**. (The virtual machine doesn't have to wait to get the possibility to run.)

Moreover, the variation between the best and the worst case is very small. Thus it can be said that **the way one virtual machine is scheduled by ESXi on the processor is very predictable.**

With four virtual machines

For this test, following virtual machines have been created:

- 3 virtual machines with Fedora Core 11. Those virtual machines are not used to perform the measurements, but to make the processor busy. Each virtual machine executes the first part of chapter 3.5.1.1.1 Overhead on CPU-Bound applications, with T = 8. That means each virtual machine executes 8 CPU-bounded threads.
- one virtual machine with Xenomai. This virtual machine is used to perform the measurements. It always wants to run, but it is in competition with the 3 other virtual machines, which also always want to run.

We already proved in chapter 3.5.1.1.1 Overhead on CPU-Bound applications that on a long time of execution, each virtual machine gets the same amount of execution time. The following test allows us to evaluate the *granularity* of ESXi's scheduler.

The worst case results are:

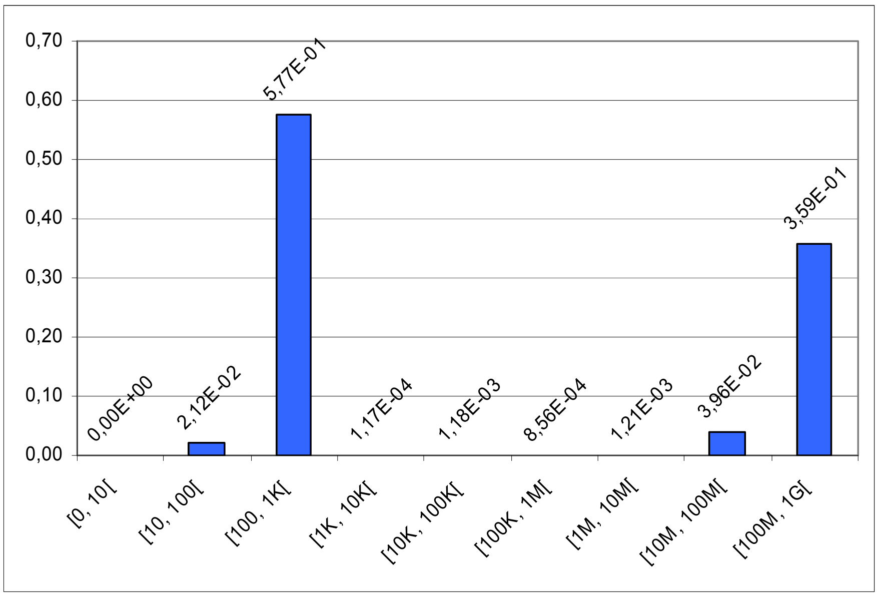


Figure 3-20: Distribution of the normalized approximate influence (4 VM)

This time, $bound_{95\%}$ = 1 billion ticks = 441 ms. It can seem huge, but actually it isn't: it is the same order of magnitude as that of a *time-slice* in a classical Linux kernel [C9]. **The ESXi's scheduler's granularity[1] has the same order of magnitude as a classical Linux scheduler's.**

Again, the variation between the best and the worst case is very small. Thus it can be said that **the way one virtual machine is scheduled by ESXi on the processor is very predictable.**

Note: This test was also performed with 4 identical virtual machines, each of them running Xenomai and measuring the ESXi scheduler's performance. All machines were synchronized as described in the second part of chapter 3.5.1.1.1 Overhead on CPU-Bound applications. A very good result was obtained, probably because each virtual machine only needs 1

[1] VMware doesn't give a lot of information about the implementation of its scheduler, but the scheduling strategy must be a sort of round-robin (time-slice scheduling). [CR1] [C7]

execution unit (8 are available on this hardware): $bound_{95\%}$ = 100 ticks = 44.1 ns. Again, the observed variation around this result is very small.

With eight virtual machines

This test is the same as the previous one, except that 7 virtual machines with Fedora are making the processor busy instead of 3. (Again, each virtual machine with Fedora runs 8 CPU-bounded threads.)

Conclusions are the same as for the previous test, and $bound_{95\%}$ = 1 billion ticks = 441 ms.

Note: Again, the test was also performed with 8 identical virtual machines. This time, it didn't make the result better: $bound_{95\%}$ = 1 billion ticks = 441 ms. Moreover:

- With only 7 identical virtual machines running, the result is still the same: $bound_{95\%}$ = 1 billion ticks = 441 ms. It may be explained as following: in this case, one execution unit is free, so the hypervisor has more "breathing space", **but** this execution unit has to share its L1 and L2 caches with the other execution unit of the same core [C16], which means it can lead to a performance loss if the free execution unit runs a different code (e.g. the code of the hypervisor).
- With only 6 identical virtual machines running, we observed a small improvement: $bound_{95\%}$ = 100 million ticks = 44.1 ms.

Again, in all those cases, the predictability was very good.

The parameter "vSphere Client / Configuration / Software / Advanced Settings / Cpu / Cpu.Quantum" probably represents the length of a time-slice for ESXi's scheduler (50 ms by default). In Comsoft's products, most threads are run with a normal priority, which means the Linux kernel gives them a time-slice of 100 ms. [C9] Distributing those threads in different virtual machines would make them alternate faster, since the time-slice given by ESXi is shorter. Thus **ESXi could be a solution** to reduce *latency* when requesting the processor and **to improve reaction time**.

Improvement attempt

Following changes have been tried, to improve the results when running 6 identical virtual machines:

- Since each virtual machine executes only one thread, the number of virtual CPUs for each virtual machine (vSphere / Edit configuration / Hardware / CPUs) has been reduced to one (instead of 8). It could have brought improvement by making the scheduling easier, but $bound_{95\%}$ fell back to 441 ms. **Reducing the number of virtual CPU per virtual machine is not a good strategy in ESXi.**
- However we left one virtual CPU per virtual machine and tried to configure the **CPU affinity** (vSphere / Edit configuration / Resources / Advanced CPU) in different profiles:
  - The first virtual machine on the first execution unit, the second machine on the second unit, etc. leaving 2 execution units of the same core free. Again, $bound_{95\%}$ = 441ms. This time, the variation between the best case and the worst case was significant: **poor predictability.**

  - With 4 virtual machines (A, B, C and D) sharing two cores, and each of both other ones (E and F) alone on one core. E and F displayed very good results: $bound_{95\%}$ = 100 ticks = 44.1 ns, whereas A, B, C and D displayed $bound_{95\%}$ = 441 ms and a **poor predictability**. This configuration doesn't fulfil our needs, because we want the same performance for each virtual machine and a good predictability.

  In conclusion, **CPU affinity mustn't be used with ESXi.**

- We set it back to eight virtual CPUs per virtual machine, and changed the parameter concerning the *hyperthreaded core sharing mode* (vSphere / Edit configuration / Resources / Advanced CPU). This parameter allows to describe how virtual CPUs (of a virtual machine) share cores with other virtual CPUs (from the same virtual machine or from others). It's described in detail in the *vSphere Administration Guide* [VV19]. This parameter had no influence on the result. $bound_{95\%}$ = 100 million ticks = 44.1 ms in all cases. So **the default value** ("Any", allowing all virtual CPUs from all virtual machines to share cores) **should be kept.**
- We set the parameter *Cpu.Quantum* (assumed to be the time-slice, see previous part) to 10 ms instead of 50 ms. No improvement has been observed: $bound_{95\%}$ = 100 million ticks = 44.1 ms. **Thus the default value should be kept**, because reducing the time-slice may reduce the overall throughput of the system [C7].

3.5.1.1.6 Conclusion

The current version of ESXi (4.0.0, Update 1, Build 219382) shows a **very good behaviour** concerning the assignment of the **microprocessor**(s) to virtual machines. This is essential because without the microprocessor, a virtual machine can't even initiate or terminate an I/O access.

During this test phase, ESXi showed **following properties**:

- low overhead on CPU-bounded applications: max. 8%;
- very good predictability during all tests;
- 20 % improvement on system calls (negative overhead);
- high overhead on MMU utilization, but limited: max. 93 %;
- same order of magnitude as Linux for the time-slice, probably even shorter.

Following parameters **have to be set**:

- VMI paravirtualization: disabled;
- CPU/MMU-Virtualization: automatic;
- number of virtual CPUs per virtual machine: maximum;
- CPU affinity: none;
- hyperthreading core sharing mode: any;
- Cpu.Quantum: 50 ms.

However, following problems have to be taken into account:

- **ESXi is unpredictable when used with two Intel Xeon E5520 microprocessors if hyperthreading is enabled.** Hyperthreading should be disabled from the BIOS for this configuration.

- **The *CPU reservation* feature can't be used if hyperthreading is enabled** (even with only one Intel Xeon E5520 processor).

Note: Those problems may be solved in the future (as you can see in the version number, this release of ESXi is very recent).

3.5.1.2 Main Memory

The second most important aspect is the main memory. It's particularly important to observe the influence of virtualization on the throughput/bandwidth and on the latency.

In contrary to what one might think, following aspects are irrelevant:

- Limitation of the amount of memory for each virtual machine: if the limit is reached, the guest operating system will use a swapping mechanism, which is not tolerated. Comsoft's current systems don't have any *swap* partition anyway, for performance reasons (when the system "swaps", it's too slow). Thus you have to guarantee that the amount of main memory needed by a virtual machine will never be higher than the available main memory. Why set a limitation if you can already guarantee that it won't be reached? We should rather pay attention to install enough main memory on the host.
- Swapping performance of the hypervisor: in case you are not able to guarantee that the total amount of main memory needed by all virtual machines is lower than the available main memory, the hypervisor also has a swapping mechanism. But as explained, no swapping is tolerated in our project.

The tests of this chapter were performed on the platform with two processors and without hyperthreading. [1]

To test the bandwidth and the latency of the main memory, test programs similar than those used in chapter 3.5.1.1 CPU Utilization are needed. The test programs have to access the main memory as often as possible. That means it has to execute the least instructions possible that don't access the main memory. It's quite a difficult programming exercise, since the following "natural" algorithm executes an already non-negligible amount of processor instructions not accessing the main memory (at least one *conditional jump*, even though recent processors are designed to optimise loops [C5]):

```
for (;;) // infinite loop
{
     temp = *mainMemoryAddress;
}
```

Moreover any modern compiler would notice that the `temp` variable is not even used and will remove the instruction inside the loop, considering it as being a good optimization. As a result the previous test program will probably not even access the main memory.

LMBench is an open-source widely-used suite of microbenchmarks for CPU, main memory, file systems, etc. [C22] In his paper Carl Staelin (from HP Laboratories) describes how powerful and extensible LMBench is. [C23] One important advantage of LMBench over other microbenchmarks is that it doesn't focus on bandwidth but measures latency too. Several advanced mechanisms are used to provide results within 1 % accuracy.

[1] This detail may be important, even when testing the main memory, because with two microprocessors, this hardware platform has a NUMA architecture (non uniform memory access): the path length from one processor to one memory slot is not constant. [C4]

The first problem mentioned (concerning the amount of processor instructions not accessing the main memory) is solved in LMBench by *loop unrolling*[1]. The second problem (concerning the compiler optimization) is solved by declaring the `temp` variable as `volatile`[2] and involving it in a (useless) computation at the end of the loop.

For more detail on LMBench, please refer Carl Staelin's article [C23], its documentation [C22] and its source code (properly documented). The document *Memory hierarchy performance measurement of commercial dual-core desktop processors* gives an example on using LMBench to perform memory performance measurements. [C24]

We used the last available version of LMBench (3.0-a9) to compare the usage of the main memory between Fedora 11 32-bits without virtualization and Fedora 11 32-bits over VMware ESXi. Again the results of those tests are biased by the imperfection already described in chapter 3.5.1.1.1 Overhead on CPU-Bound applications (other small tasks are running, the system is disturbed by interrupts, etc.), but again we can reasonably consider that those effects are negligible. Moreover the recommendations displayed during the LMBench installation (e.g. the X server should be turned off) have been respected.

We made a default configuration of LMBench during the installation process. Without virtualization, the installation process indicated a *timing granularity* of 50 ms after some measurements, which means it considers that running each test during only 50 ms will be enough to reach a good precision. Each test will be run 11 times and the median result will be delivered. [C23] On the system over ESXi, the *timing granularity* is 10 ms.

All scripts written to automate the tests and display the results are available on the CD-ROM: `/eval/ram/` .

3.5.1.2.1 Memory Bandwidth

Without virtualization

This test uses the *bw_mem* microbenchmark. It measures the bandwidth when reading from or writing into an array. [C25] The array size can be specified and this test has been run five times for several array sizes. To run this test yourself, type one of the following commands:

```
$ ./run.sh bw_rd_profile
$ ./run.sh bw_wr_profile
```

To display the results, type one of the following commands:

```
$ ./display.sh bw_rd_profile
$ ./display.sh bw_wr_profile
```

[1] Loop unrolling is the replacement of a loop by a sequence of identical iterations. [C4]
[2] `volatile` is a C++ keyword telling the compiler not to make any assumption concerning the value of a variable, but to act as if its value could be modified at any time by an external agent.

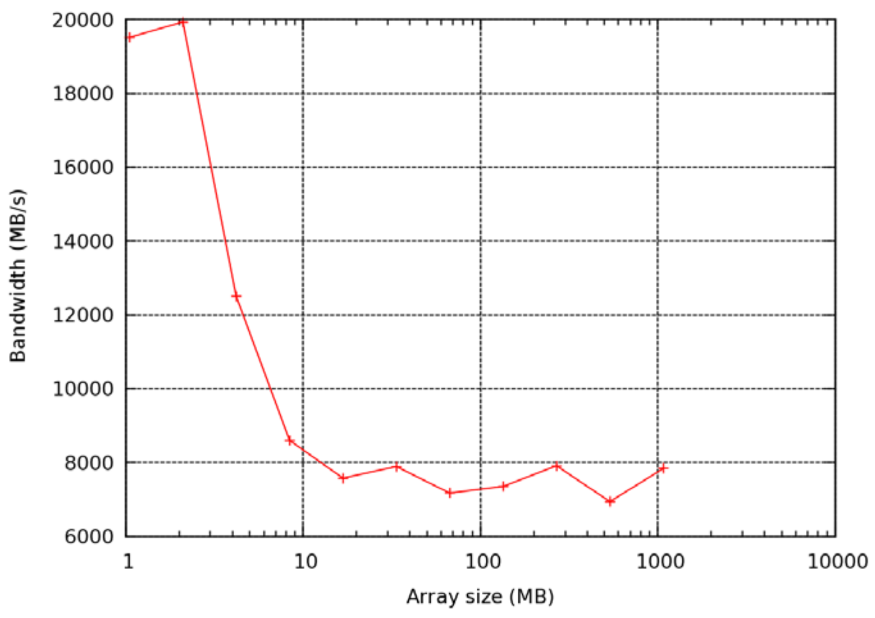


Figure 3-21: Read-bandwidth without virtualization

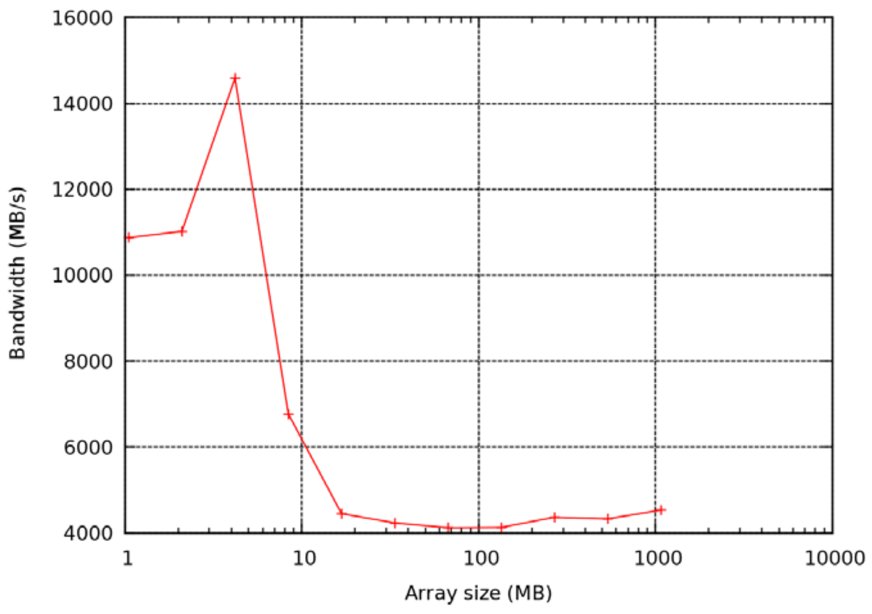


Figure 3-22: Write-bandwidth without virtualization

Figures 3-21 and 3-22 show the results. As we can see, the bandwidth is higher for array sizes shorter than 8 MB since the array fits in the L3 cache (see chapter 3.5 Evaluation). For larger arrays the test program actually accesses the main memory:

- Mean read-bandwidth: about 7400 MB/s;
- Mean write-bandwidth: about 4200 MB/s.

We tried to run the same test again while disturbing it with 6 other processes doing exactly the same (also running *bw_mem*). To run this test yourself, type one of both following commands:

```
$ ./run.sh bw_rd_disturb_profile
$ ./run.sh bw_wr_disturb_profile
```

To display the results, type one of both following commands:

```
$ ./display.sh bw_rd_disturb_profile
$ ./display.sh bw_wr_disturb_profile
```

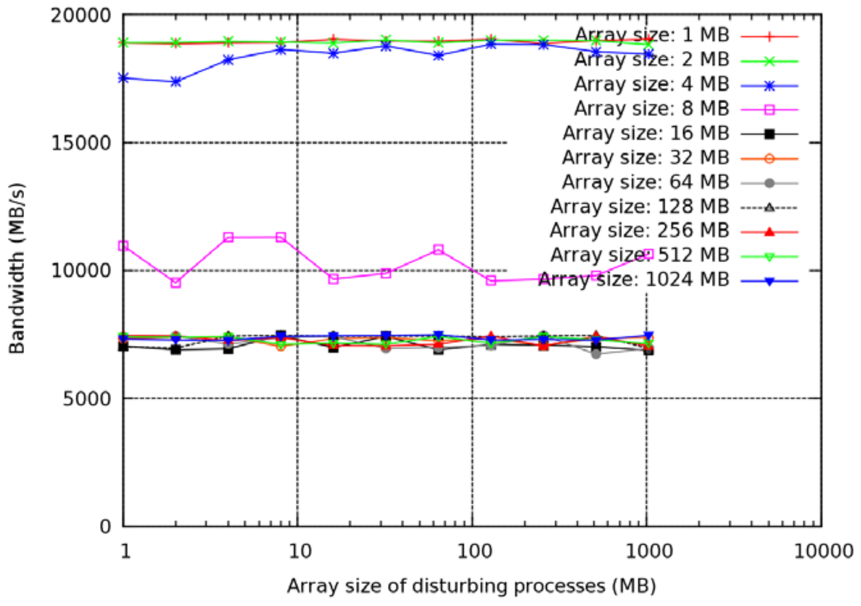


Figure 3-23: Disturbed read-bandwidth without virtualization

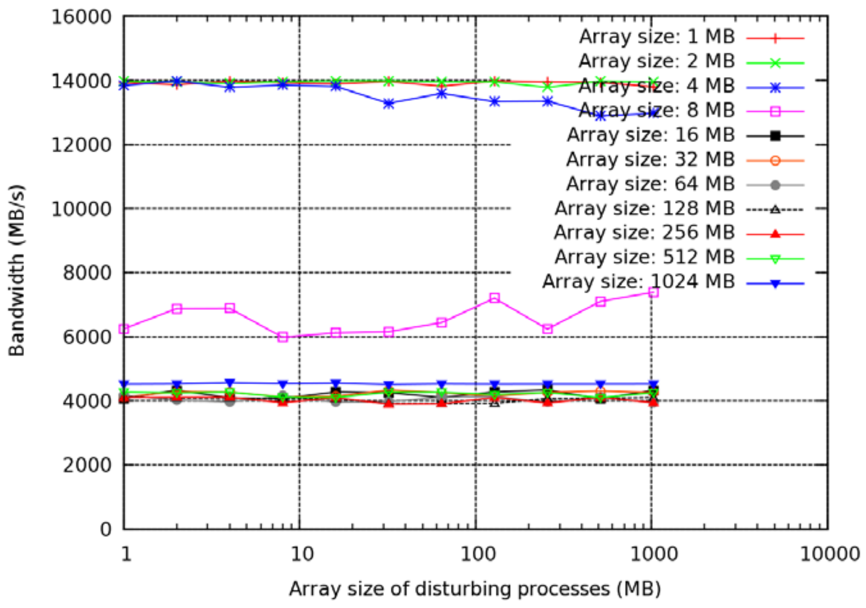


Figure 3-24: Disturbed write-bandwidth without virtualization

Figures 3-23 and 3-24 show the results. As we can see, the results are the same: it didn't reduce the bandwidth, whereas 7 execution units of 8 were accessing the most often possible the main memory. Actually the write-bandwidth with an array size of 1 MB for example has even been improved, probably due to some synergetic effects. [C6]

We can already conclude that **on this architecture the access to the memory hierarchy** (caches + main memory) **is not a bottleneck.**

Note: No more than seven instances of the test program should be run at the same time, otherwise the results will be skewed by the scheduling and won't mean anything. The processor is needed to initiate any access to any resource, therefore you need it during the whole test to measure the access to the tested resource (here the main memory).

Over ESXi

The exact same sequence of tests was run in a virtual machine. Each disturbing instance of the test program was run in its own virtual machine (which means seven virtual machines were needed to run the second part of those tests). Therefore the commands

```
$ ./run.sh bw_rd_disturb_profile
$ ./run.sh bw_wr_disturb_profile
```

can no longer be used. Each disturbing instance has to be launched in a separate virtual machine by means of one of these commands:

```
$ ./busyram.sh <size> rd
$ ./busyram.sh <size> wr
```

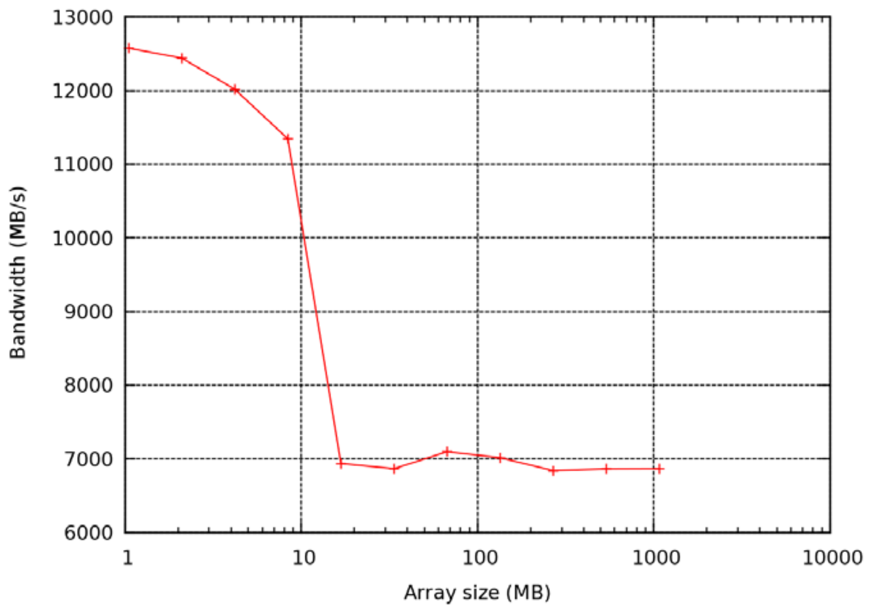


Figure 3-25: Read-bandwidth over ESXi

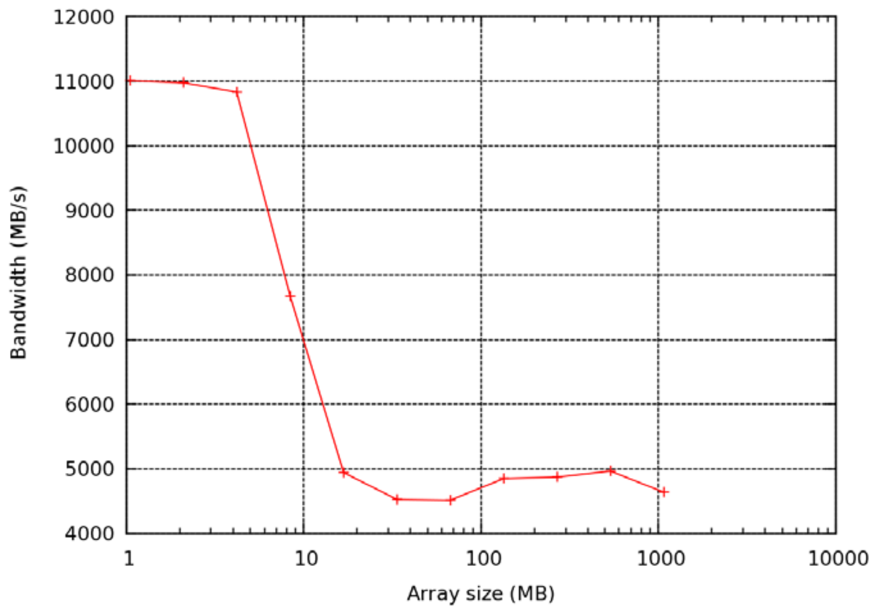


Figure 3-26: Write-bandwidth over ESXi

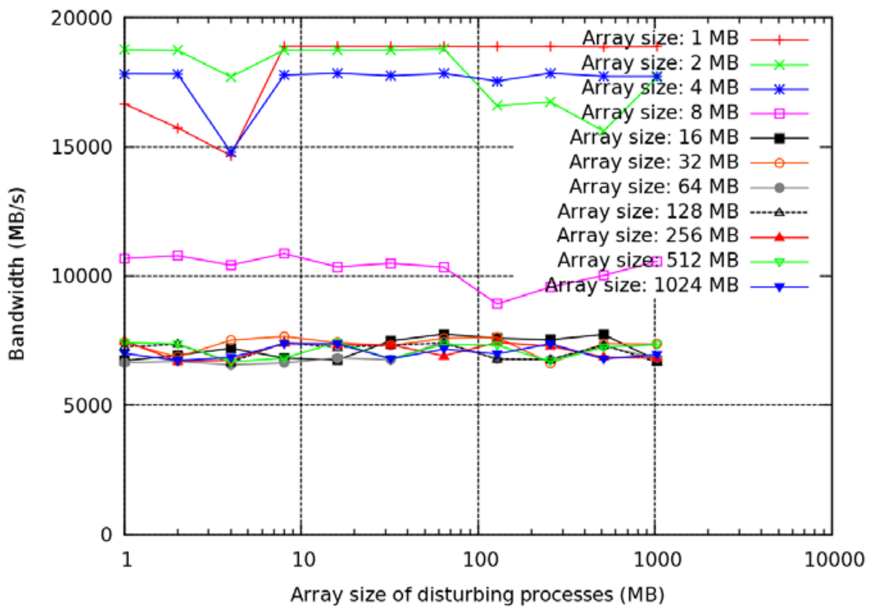


Figure 3-27: Disturbed read-bandwidth over ESXi

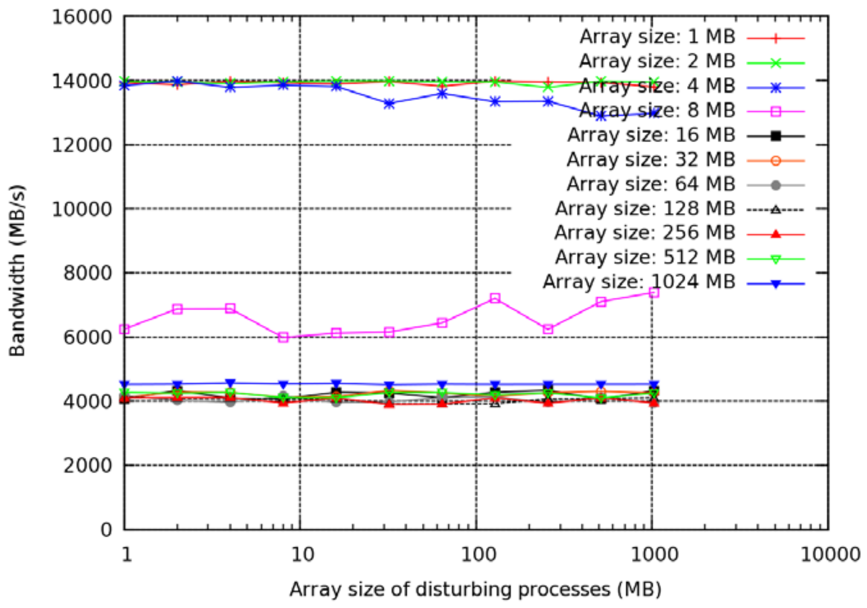


Figure 3-28: Disturbed write-bandwidth over ESXi

Figures 3-25 and 3-26 show a loss of performance concerning the caches but no loss concerning the main memory:

- Mean read-bandwidth: about 7000 MB/s (5 % overhead);
- Mean write-bandwidth: about 4700 MB/s (12 % improvement).

Figures 3-27 and 3-28 show that the cache bandwidth can be slightly disturbed but again, the bandwidth when accessing the main memory is not reduced by the presence of the disturbing instances of the test program.

ESXi shows native performances concerning the memory bandwidth.

3.5.1.2.2 Memory Latency

Without virtualization

This test uses the *lat_mem_rd* microbenchmark. It first allocates an array and then performs a latency test. The test is run for several array sizes (X-axis). The latency test works as follows: the microbenchmark starts at the end of the array, and reads backwards through the array. It is implemented in such a manner that the array accesses are dependant on one another, forcing the processor to really read the array (no optimization is possible). [C25] The duration of one test is the *timing granularity* (50 ms in this case, see chapter 3.5.1.2 Main Memory). If the microbenchmark is able to perform one million *loads* during those 50 ms, then the mean *load duration* (or mean latency over this short period of 50 ms) is 50 ns (Y-axis).

The microbenchmark considers the allocated array as an array of bytes. It doesn't necessarily read *every* byte backwards but actually every Nth byte. N is called the *stride size*. With a stride size of 1, every byte of the array is read. The test has been run with different stride sizes (several curves).

Varying the array size helps to determine the different cache size. Each curve of the figure 3-29 for example has four plateaus:

- The first one ends with an array size of 32 kB, indicating that any array smaller than 32 kB fits in the L1 data cache, offering a small latency of 1.7 ns. The size of the L1 data cache is probably 32 kB. This is actually the case (see chapter 3.5 Evaluation).
- The second plateau indicates a 256 kB long L2 data cache, where the array is too long to fit in the L1 data cache but short enough to fit in the L2 data cache.
- The third plateau indicates an 8 MB long L3 cache.
- The last plateau represents the main memory latency.

Varying the stride size helps to determine the cache line size. For example the last plateau in figure 3-29 is more or less high (different latencies) depending on the stride size. A L3 cache line size of 256 bytes would be an explanation: when reading a byte in the main memory, a block of 256 bytes is actually fetched from the main memory and copied into a cache line of the L3 cache. If the stride size is bigger than or equal to 256 bytes, a block has to be fetched on each byte to be read. If the stride size is 128 bytes, one fetched block delivers two "interesting bytes", thus reducing the amount of main memory accesses and the mean latency displayed on the curve. We can conclude that:

- the L1 cache line size is 64 bytes;
- the L2 cache line size is 256 bytes;
- the L3 cache line size is 256 bytes.

Of course the most interesting values for this chapter are the values where the caches are not involved: i.e. the last plateau (corresponding to the main memory) where stride size is greater than or equal to 256 bytes.

To run this test yourself and display the results, type the following commands:

```
$ ./run.sh lat_rd_profile
$ ./display.sh lat_rd_profile
```

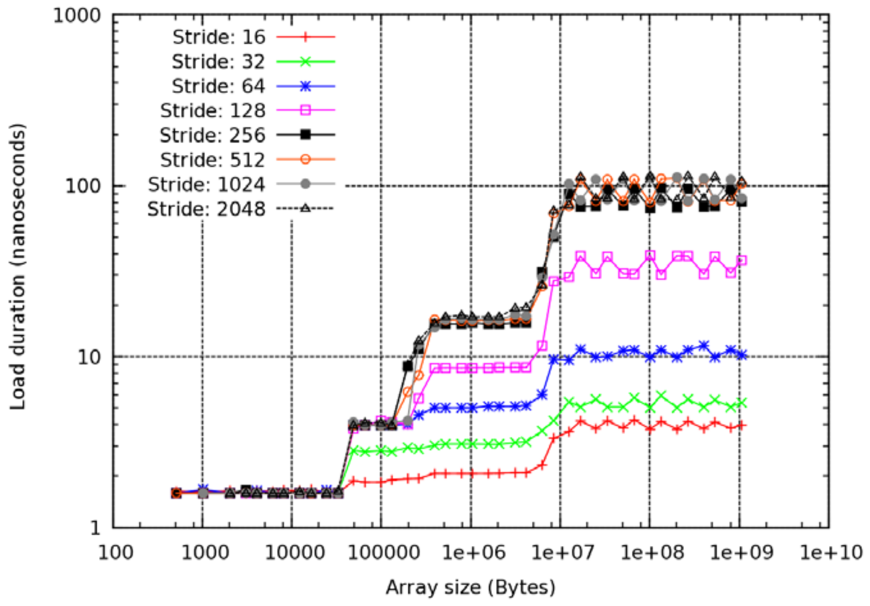


Figure 3-29: Read-latency without virtualization

Figure 3-29 shows the results. As we can see, the latency when reading a byte from the main memory is about 100 ns.

We tried to run the same test again while disturbing it with 6 other processes running *bw_mem* the whole time, to access the main memory as often as possible and disturb the process measuring the latency. We didn't expect any difference in the result, since we prove in the memory bandwidth test (see chapter 3.5.1.2.1 Memory Bandwidth) that memory access is not a bottleneck. To run this test yourself and display the results, type the following commands:

```
$ ./run.sh lat_rd_disturb_profile
$ ./display.sh lat_rd_disturb_profile
```

The obtained curves were almost the same as without perturbation.

Over ESXi

The exact same sequence of tests was run in a virtual machine. Each disturbing instance of the test program was run in its own virtual machine (which means seven virtual machines were needed to run the second part of those tests). Therefore the command

```
$ ./run.sh lat_rd_disturb_profile
```

can no longer be used. Each disturbing instance has to be launched in a separate virtual machine by means of the following command:

```
$ ./busyram.sh 1024m rd
```

The obtained curves were almost identical as without virtualization. **ESXi shows native performances concerning the memory latency.**

3.5.1.2.3 Conclusion

ESXi shows native performances concerning memory bandwidth and latency. They are **constant**, **predictable** and independent of the number of running threads or virtual machines.

3.5.1.3 Hard Disk Drive

In order to test the performances when accessing the hard disk drive, a test program has been developed. This test program measures the throughput/bandwidth and the latency when writing into a file (on a hard disk drive), with a precision of 50 ms. To achieve this, the program periodically appends a "block" (e.g. 2 bytes) at the end of the output file and it can be detected if the program "freezes" more than 50 ms when trying to access the file.

Here is its algorithm: (the full implementation is available on the CD-ROM: `/eval/hdd/write/`)

```
thread // statistic thread
{
    latency = 0;
    do every 50 ms during 30 seconds
    {
        if (elapsedTime % 1000 ms == 0) // every second
        {
            perform and save throughput over the last second in RAM;
        }
```

```
          if (elapsedTime % 200 ms == 0) // every 200 ms
          {
               perform and save throughput over the last 200 ms in RAM;
          }

          // every 50 ms:
          perform and save throughput over the last 50 ms in RAM;
          if (throughput == 0)
          {
               latency += 50 ms;
          }
          else
          {
               save latency in RAM;
               latency = 0;
          }
     }
     make main thread exit its infinite loop;
}

thread // main thread
{
     create and open the file;
     create statistic thread;

     for (;;) // infinite loop
     {
          write(block, file);
          if (option –l) flush user-space buffer;
          if (option –s) force write on the device;

          send information to statistic thread so that it can perform
               throughput and latency measurement;
     }

     display mean throughput;
     display Bound_95%;
     save results of statistic thread into a file;
}
```

A standard program (like this one) doesn't directly access the hard disk drive when writing:

- The `write()` function is provided by a library. To reduce the number of switches between user-mode and kernel-mode, this library creates a buffer (one per application): the user-space buffer. When calling `write()`, the buffer is just filled, and the function returns. When this buffer is full enough, the next call to `write()` will actually trigger a switch to kernel-mode and the kernel will take all data from this buffer.
- The kernel doesn't directly write on the hard disk drive. It fills on its turn its own buffer (one per hard disk drive). When the buffer is full enough or after a certain timeout (it depends on the hard disk scheduling strategy), it sends all the data to the hard disk drive in a particular order, to optimize the movements of the hard disk and to reach a high throughput.

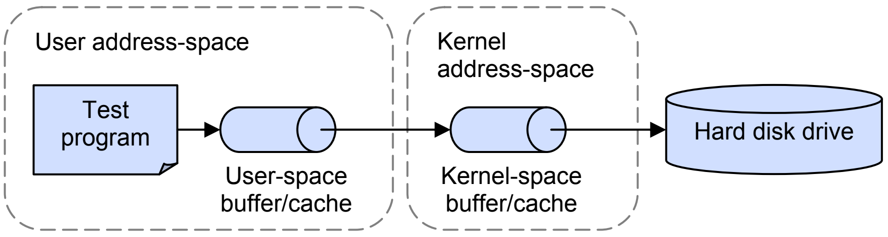


Figure 3-30: Path from application to hard disk drive

More information on disk scheduling and buffering can be found in *Operating Systems* [C7].

Note: On figure 3-30, the *hard disk drive* is just an abstraction because:

- nowadays, most hard disk drives have on their turn one or several caches to optimize write and read operations themselves;
- the RAID-controllers of our platform have a cache, again for the same purpose (see chapter 3.5 Evaluation).

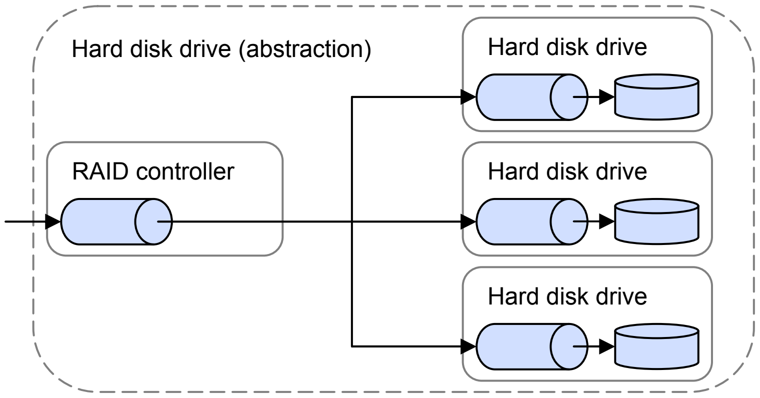


Figure 3-31: Hard disk drive abstraction

I/O-Scheduler und RAID-Performance [C26] is a very interesting article on the influence of RAID on disk scheduling.

The kernel normally uses *Direct Memory Access* (DMA), a technique allowing a device to access directly the main memory in order to perform the whole transfer without the need of the CPU (the CPU is needed to start and to terminate the transfer). This technique is usually problematic with virtualization, because when starting the transfer, the *virtual* CPU gives an address from the *virtual* main memory to the *virtual* device, making the use of DMA (which is a *physical* optimization) very difficult. Intel VT-d offers a hardware support for this purpose [V5] and our test program will allow us to measure its performance when used by ESXi.

The test program has different options (type `./hdd -h` to get the complete list). The most important are:

- `-k`: disables kernel buffering (no kernel-space buffer will be used);
- `-u`: disables user-level buffering (no user-space buffer will be used);
- `-l`: flushes the user-level buffer after each written block (data is forced to leave the user-space buffer, if any);

- `-s`: forces physical write after each block (data is forced to leave both user-space and kernel-space buffers);[1]
- `-b`: specifies the size of the blocks (in bytes) to be written into the file on each iteration.

This test program has been run in the following context:

- the file is created on an *ext3* partition of 100 GB with a block size of 4096 bytes;
- the *ext3* partition was created on a RAID 5 group based on 4 hard disk drives connected to the same hardware *RAID channel* (see figure 3-4). This partition was dedicated to the test: the operating system, the virtualization layer if any and the test program were located on another RAID group connected to the other *RAID controller* (see figure 3-4) to avoid as many side effects as possible.

The different options can be combined together and after some tests, their compatibility with each other has been determined:

When the option -k is used, you cannot use the -u option.

When the option -l is used, using the option -u has no influence on the result.

	-k	-u	-l	-s	No option
-k		x	√	√	√
-u	x		=	√	√
-l	√	√		√	√
-s	√	√	√		√

When the option -k is used, using the option -s or not influences the result.

Table 3-14: Compatibility and dependency between the options

Tests showed that the options `-u` and `-l` are equivalent when the block size (specified with the option `-b`) is small (a few bytes), whereas `-u` alone has no effect when the block size is bigger (a few kilobytes). Therefore `-l` is more efficient.

In this context, `-kl` only works when the block size is a multiple of 512 bytes (**device**'s block size).[2]

Note: We used RAID 5 because it's the method usually used by Comsoft to ensure data redundancy.

[1] The implementation uses the `fsync()` function. It isn't clear in the documentation if this function forces the data to leave the hardware buffers too. It seems to depend on the implementation of the driver.

[2] For information, the implementation of the option `-k` uses the `O_DIRECT` flag of the `open()` function.

3.5.1.3.1 Maximum throughput on a long execution time

Without virtualization

Of course the maximum throughput is reached when all optimizations are enabled (no option used for the test program) and when the block size is big enough. We reached it with a block size greater than or equal to 64 bytes. You can run the test yourself and display the results with the following commands:

```
$ ./hdd -b 64 -f <filepath_leading_to_the_test_partition>
$ ./display.sh throughput
```

Again, like for the previous tests (CPU and main memory tests), we consider this test to be precise enough, since the load of the CPU when the test program is not running is lower than 1 %. Moreover, the time precision we are dealing with is much lower than for the previous tests (milliseconds instead of nanoseconds).

The test was run several times over Fedora 11 without virtualization layer. Figure 3-32 shows the results of one execution:

- The yellow curve represents the measurement of the throughput with a period of 50 ms.
- The red one with a period of 200 ms.
- The blue one with a period of one second.

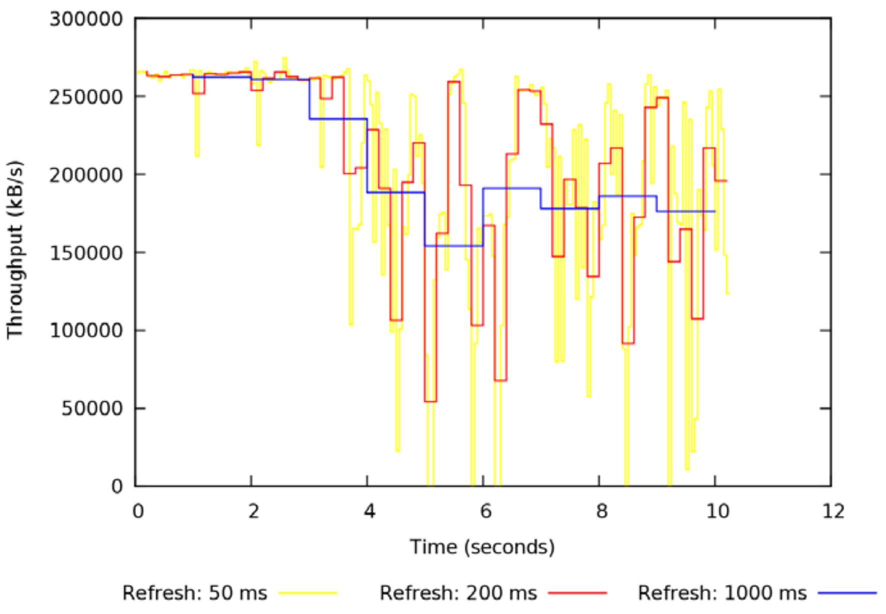


Figure 3-32: Disk throughput without virtualization

As we can see, the throughput is very high at the beginning: the computer doesn't access the hard disk at this time but is just filling the buffers located in the main memory. After less than four seconds, the operating system considers the buffers to be full and starts to access the hard disk drive or, at least, to communicate with the RAID controller. After ten seconds, the program stops because the file reached its maximum size of 2 GB.

We are sure that the operating system is sending the data to the RAID controller during the last six seconds because the main memory allocation stops growing (visible with the `htop` command) and is smaller than 1 GB. Moreover no *execution unit* is fully used during the execution, confirming that the RAID group is well and truly the bottleneck in this transfer.

During the last six seconds, the throughput varies around 175 MB/s. The mean throughput for the whole exchange is about 200 MB/s (varying between 190 and 210 MB/s).

Over ESXi

The same test was run over ESXi. We then have an additional layer: a *virtual* hard disk drive have been created on the real one, by means of the vSphere Client. This operation actually creates a VMFS partition (Virtual Machine File System) on the physical hard disk. The default parameters have been used:

- Max. file size: 256 GB, block size: 1 MB;
- VMFS version: 3.33

The virtual hard disk has a size of 100 GB and an *ext3* partition similar to the one used for the previous test was created with the guest operating system, taking up all of the available space.

When creating the virtual hard disk drive, several types of (virtual) SCSI controllers are available: [VV19]

- LSI Logic SAS;
- Bus Logic Parallel;
- LSI Logic Parallel (by default);
- Paravirtualization.

At first glance, LSI Logic SAS seems more adapted to our situation because we have SAS hard disk drives (Serial Attached SCSI).

Again, the test was run several times with a block size of 64 bytes. Here are the results:

Controller type	Throughput after 4 seconds	Note
LSI Logic SAS	≈ 150 MB/s (85 %)	See figure 3-33. Increasing the block size doesn't improve the result.
Bus Logic Parallel	≈ 7 MB/s (4 %)	Increasing the block size doesn't improve the result.
LSI Logic Parallel	≈ 150 MB/s (85 %)	Increasing the block size doesn't improve the result.
Paravirtualization	≈ 150 MB/s (85 %)	Increasing the block size doesn't improve the result.

Table 3-15: Disk throughput over ESXi

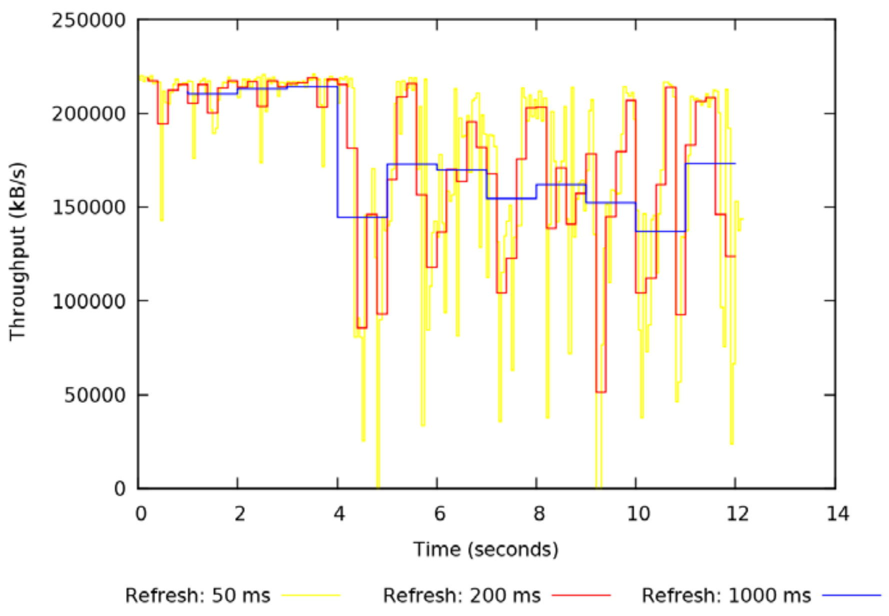


Figure 3-33: Disk throughput with LSI Logic SAS virtual controller

The controller type **Bus Logic Parallel** can already be **excluded**, because of its poor performance.

We also **exclude** the controller type **Paravirtualization**, because the guest operating system can't be used on such a virtual drive: a kernel module (*pvscsi*) has to be loaded into the guest kernel so that the virtual drive can be used. Therefore a virtual machine can't boot on this virtual drive because the guest kernel is not started yet. We need predictability for **all** communications with the hard disk drives and thus we have to find a solution valid for the drive where the guest operating system is installed too. More details on this controller type are available in the document *Configuring disks to use VMware Paravirtual SCSI adapters* [VV24].

Other controller types showed **near-native performance**.

3.5.1.3.2 Throughput with several instances of the test program

Without virtualization

The same test program was run, while other instances of the program were doing the same on the same partition. We expect the throughput of each instance to be divided by the number of instances. Only one instance gathers statistics. The "disturbing" instances are launched silently inside a loop with the following command:

```
$ ./loop.sh –n –f <file> -b 64
```

The file name must be different for all instances. Here are the results without virtualization:

Number of instances	Throughput
2 (one disturbing instance)	83.8 MB/s (σ_X = 1.2 MB/s), 47.8 % (of 175 MB/s)
4	41.4 MB/s (σ_X = 0.6 MB/s), 23.7 %
8	18.8 MB/s (σ_X = 0.9 MB/s), 10.7 %

Table 3-16: Disk throughput without virtualization

As we can see, the throughput scales well with the number of instances. We didn't observe any "warm-up phase" during the first seconds. Increasing the block size of **all** instances doesn't change the result. Though increasing block size of one instance gives it a small advantage on the others (slightly higher throughput).

Over ESXi

We did the same test over ESXi. First we launched all instances in the same virtual machine:

Controller type	Number of instances	Throughput
LSI Logic SAS	2 (one disturbing instance)	76.6 MB/s (σ_X = 4.7 MB/s), 51.1 % (of 150 MB/s)
	4	36.8 MB/s (σ_X = 2.2 MB/s), 24.5 %
	8	18.8 MB/s (σ_X = 1.0 MB/s), 12.5 %
LSI Logic Parallel	2	69.2 MB/s (σ_X = 2.3 MB/s), 46.1 %
	4	34.4 MB/s (σ_X = 0.7 MB/s), 22.9 %
	8	16.3 MB/s (σ_X = 2.4 MB/s), 10.9 %

Table 3-17: Disk throughput over ESXi

The conclusions are the same as without virtualization: the throughput scales well with the number of instances. We didn't observe any "warm-up phase" during the first seconds. Increasing the block size of **all** instances doesn't change the result. Though increasing block size of one instance gives it a small advantage on the others (slightly higher throughput).

Note that the standard deviation is a little bit higher than without virtualization but it stays reasonable. For example in the first line of the table 3-17 the deviation of 4.7 MB/s represents only 6 % of the throughput. Moreover a hard disk drive is not a device for which you can expect a low deviation because of the underlying (mechanical) technology.

For the following test, we put the "disturbing" instances of the program in a second virtual machine. We expect ESXi to assign 50 % of the disk throughput to the measuring instance (alone in the first virtual machine) and to share the rest of the throughput among the disturbing instances (all in the second virtual machine).

To do so, we created two virtual hard disk drives located on the tested RAID 5 group and assigned each virtual drive to one virtual machine. The virtual machines were using the same type of SCSI controller: we didn't mix because the result wouldn't be interpretable:

Controller type	Number of instances	Throughput
LSI Logic SAS	2 (one disturbing instance)	68.8 MB/s (σ_X = 6.8 MB/s), 45.9 % (of 150 MB/s)
	4	72.5 MB/s (σ_X = 8.3 MB/s), 48.3 %
	8	64.7 MB/s (σ_X = 3.7 MB/s), 43.1 %
LSI Logic Parallel	2	59.7 MB/s (σ_X = 8.0 MB/s), 39.8 %
	4	45.4 MB/s (σ_X = 8.6 MB/s), 30.3 %
	8	38.5 MB/s (σ_X = 3.0 MB/s), 25.7 %

Table 3-18: Throughput sharing among virtual machines

As we can see, **LSI Logic SAS** really makes a **good isolation** between virtual machines by sharing throughput based on virtual machines instead of processes: in this experiment, each machine is given 50 % of the throughput, independently from the number of processes trying to access the hard disk drive.

Again, the deviation is higher than without virtualization but stays reasonable. Virtual machines using LSI Logic SAS as a virtual SCSI controller seem to be a **good solution to isolate applications from each other**. But we still need to analyse the latency of this solution when accessing the hard disk drive (see next chapter).

The isolation provided by LSI Logic Parallel is not so good. We could have excluded this controller type now but we decided to evaluate its guarantees concerning latency.

3.5.1.3.3 Latency

As explained, our test program is able to detect latencies of over 50 ms. This latency is the time between the moment when the program sends the block of data and the moment when the program can continue its execution (and send the next block). Between those two moments, the execution of the program is blocked.

It delivers a graph and computes the *bound*$_{95\%}$ indicator, presented in chapter 3.5.1.1.5 Scheduling. Briefly, bound$_{95\%}$ is the maximum observed latency, except if the number of occurrences of this latency is so low that it represents less than 5 % of the total execution time.

Consider for example the figure 3-34 representing in a histogram the detected latency during the execution of the program (it's actually the worst case because we executed the test program several times). Note that latencies shorter than 50 ms were not detected and therefore are not represented on the graph. The execution time of the test program was exactly 12.15 seconds. A latency to access the hard disk between 550 and 600 ms has been observed once. To simplify, we consider that this access lasted 575 ms. But 575 ms represents only 4.7 % of 12.15 seconds (this percentage is called *influence*, see chapter 3.5.1.1.5 Scheduling). We can ignore it, because it represents less than 5 %. Then a latency of 275 ms to access the hard disk has been observed once, representing 2.3 % of the execution time. We have to add to it the *influences* of all higher latencies:

2.3 % + 4.7 % = 7 % > 5 %

It can't be ignored anymore and the program concludes:

- maximum latency = 600 ms;
- bound$_{95\%}$ = 300 ms.

Note: the execution of the program is limited to 30 seconds (it stops when the file size reaches 2 GB or after 30 seconds, whatever comes first). 5 % of 30 seconds is 1.5 second, which means that as soon as a latency longer than or equal to 1.5 second has been detected, $\text{bound}_{95\%}$ = maximum latency. This is why this indicator can't hide latencies greater than 1.5 seconds (which is good, since it corresponds to the time granularity of Comsoft's soft real-time products, see chapter 2.1.1 Comsoft products) and is a fair indicator to compare solutions where maximum latencies are lower than 1.5 second. Briefly: **the highest latency will be ignored if it happens rarely, except if it's greater than 1.5 second.**

Without virtualization

Once again, the same test program has been used, with all optimizations (without any option) and with a block size of 64 bytes. You can run the test yourself and display the results with the following commands:

```
$ ./hdd -b 64 -f <filepath_leading_to_the_test_partition>
$ ./display.sh latency
```

The worst case observed is **$\text{bound}_{95\%}$ = 50 ms** (with a maximum latency of 150 ms).

Over ESXi

The same test has been done in a virtual machine, to measure the latency induced by ESXi. **LSI Logic SAS** showed **native performance**, whereas LSI Logic Parallel showed $\text{bound}_{95\%}$ = 300 ms, with a maximum latency of 600 ms:

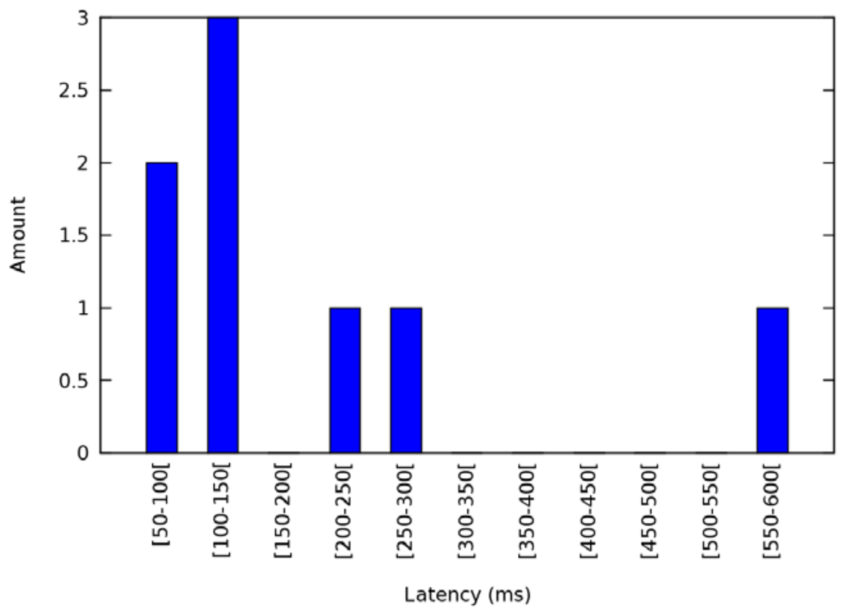


Figure 3-34: Disk latency with LSI Logic Parallel virtual controller

Therefore we decided to **exclude LSI Logic Parallel**, given the mediocre results for both latency and throughput isolation among virtual machines.

3.5.1.3.4 Latency with several instances of the test program

For now and unless otherwise specified, the LSI Logic SAS virtual controller will be used.

We now come to the **main point of this thesis: the latency when several applications need to access the same hard disk drive.**

One of the best ways we found to precisely quantify this latency is to launch the following instances of the test program:

- One *measuring instance*, gathering statistics. This instance should measure the time needed to really access the device (not only the buffers located in the main memory) and only to access it (without even trying to transfer a huge amount of data). Therefore the test program will transfer blocks of only one byte, force the data to leave the buffers after each transferred byte, and disable the kernel-space buffer:

```
$ ./hdd -ks -b 1 -f <file>
```

- Several *disturbing instances*, communicating the most possible with the device and disturbing the most possible the *measuring instance*. We already know that the highest throughput is reached when all optimizations are enabled and when the block size is greater than or equal to 64 bytes. We discovered that increasing this block size disturbs the measuring instance even more, and that a maximum is reached with a block size greater or equal to one kilobyte:

```
$ ./loop.sh -b 1024 -f <file>
```

Without virtualization

We ran the tests over Fedora 11 without virtualization. The following table shows the results (worst cases observed):

Number of disturbing instances	$\text{bound}_{95\%}$	Throughput of the measuring instance (for information)
0	No latency detected	350 kB/s
1	1.7 s	35 kB/s
3	2.1 s	6 kB/s
7	3.1 s	5 kB/s

Table 3-19: Disk latency without virtualization

Note: Do not compare the throughput of the measuring instance with the throughput observed in the previous tests! The throughput showed in the previous table is lower because we disabled a lot of optimizations and the transferred blocks are smaller.

As we can see, the observed latency is **very high** (an application needing to send one byte to the hard disk drive may have to wait for 3.1 s) and we need ESXi to reduce it.

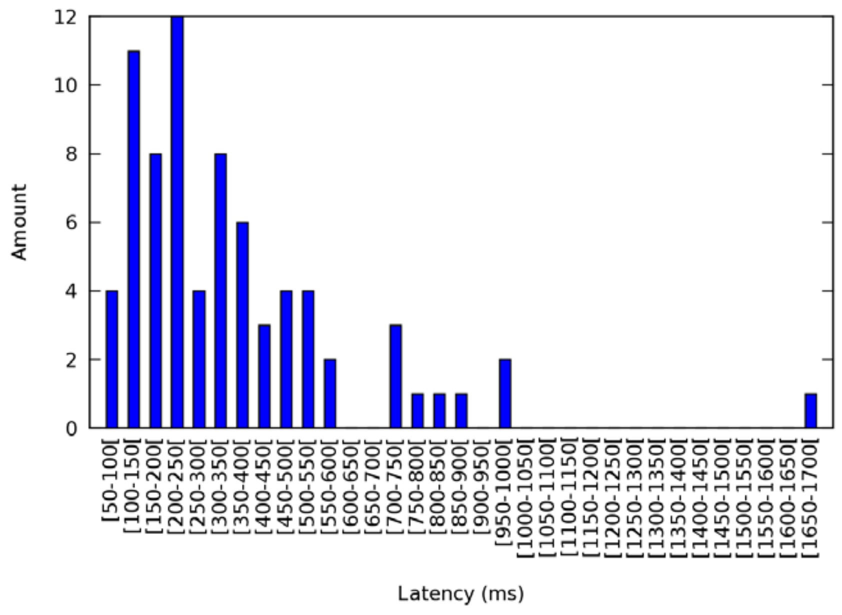


Figure 3-35: Latency with one disturbing instance (without virtualization)

Over ESXi

The test has been run over ESXi, with the following configuration:

- the measuring instance is in one virtual machine;
- the disturbing instances are together in a second virtual machine.

We expect from the hypervisor a reduction of $\text{bound}_{95\%}$ to an acceptable level. The following table shows the results (worst cases observed):

Number of disturbing instances	$\text{bound}_{95\%}$	Throughput of the measuring instance (for information)
0	No latency detected	263 kB/s
1	400 ms	122 kB/s
3	2.0 s	96 kB/s
7	2.0 s	96 kB/s

Table 3-20: Disk latency over ESXi

Again we can see that a good isolation between virtual machines is ensured concerning the throughput but the results concerning the latency are still **not satisfying** although slightly better.

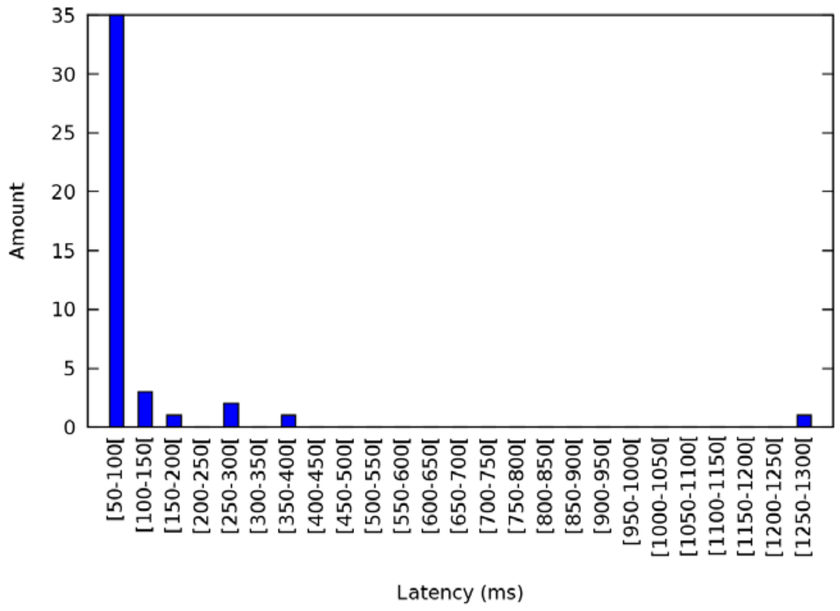
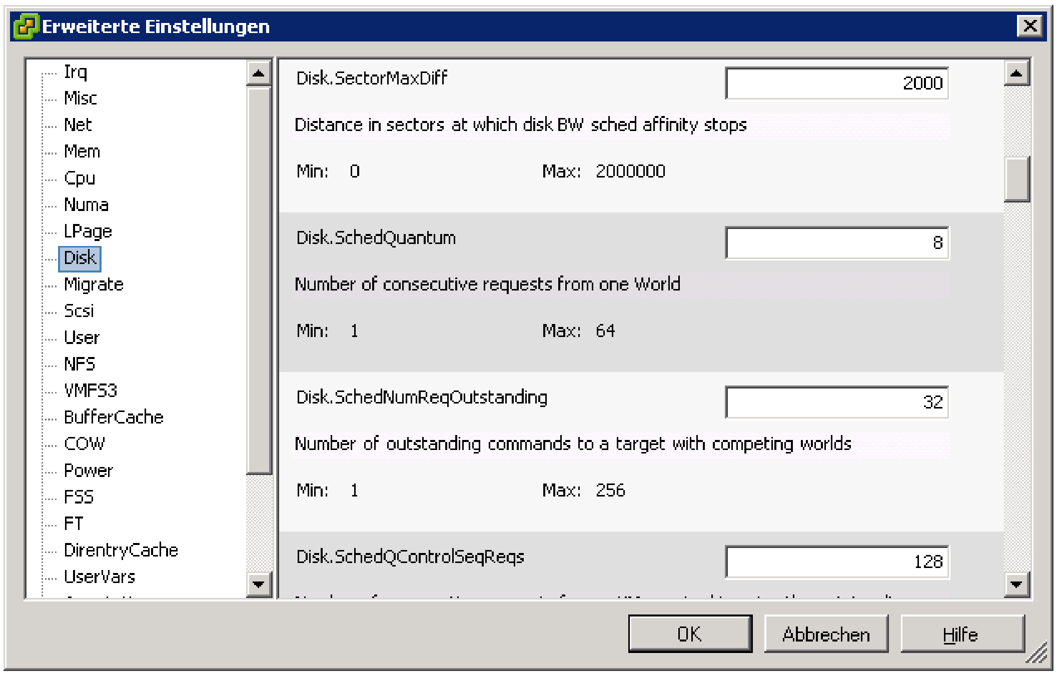


Figure 3-36: Latency with one disturbing instance (over ESXi)

Note: ESXi also provides the possibility to link a virtual hard disk drive to a virtual *IDE controller*, but it didn't reduce the latency.

3.5.1.3.5 Improvement attempt

ESXi provides a long list of parameters that can be changed. Those parameters can be found in vSphere Client / Host configuration / Software / Advanced configuration. They are **not documented**. They are sorted into sections (Cpu, Disk, Scsi, ...) and each parameter comes with a short description:



Disk.SectorMaxDiff 2000
Distance in sectors at which disk BW sched affinity stops
Min: 0 Max: 2000000

Disk.SchedQuantum 8
Number of consecutive requests from one World
Min: 1 Max: 64

Disk.SchedNumReqOutstanding 32
Number of outstanding commands to a target with competing worlds
Min: 1 Max: 256

Disk.SchedQControlSeqReqs 128

Figure 3-37: Advanced parameters in vSphere Client

We analysed the description of each parameter in the sections *Disk* and *Scsi* and tried to reduce or to increase the value of the following parameters:

Reduced parameters	Increased parameters
`Disk.SectorMaxDiff`	`Disk.SchedQControlSeqReqs`
`Disk.SchedQuantum`	`Disk.QFullSampleSize`
`Disk.SchedNumReqOutstanding`	
`Disk.SchedQControlVMSwitches`	
`Disk.DelayOnBusy`	
`Disk.ResetLatency`	
`Disk.MaxResetLatency`	
`Disk.ResetPeriod`	
`Disk.DiskMaxIOSize`	

Table 3-21: Parameters changed to reduce disk access latency

We tried several combinations but it never had any effect on the latency, even after rebooting the host. (Some parameter changes only take effect after a reboot of the host platform)

Then we decided to change the parameter `Disk.BandwidthCap`. This parameter is a limitation of the disk throughput/bandwidth (in kB/s) for all virtual machines. There is no way to set a different limitation for each virtual machine. The default and maximum value of this parameter is 2^{32}-2 kB/s ≈ 4 TB/s, in other words: *no limitation*. Setting a limitation and restarting the host (needed) **improves immediately the results**:

Disk.BandwidthCap	Number of disturbing instances	$\text{bound}_{95\%}$	Throughput of the measuring instance (for information)
130 MB/s (87 % of 150 MB/s)	1	100 ms	80 kB/s
	3	250 ms	65 kB/s
	7	350 ms	65 kB/s

Table 3-22: Disk latency over ESXi with limitation of the disk bandwidth

As you can see, limiting all virtual machines to 87 % of its maximal throughput and therefore limiting the disturbing virtual machine allows the measuring virtual machine to access the hard disk drive without being disturbed. It stops working if the limitation is greater than 87 %.

We ran another test with the following instances:

- one virtual machine running only one measuring instance;
- three virtual machines, each of them running eight disturbing instances.

We obtained a **good result: $\text{bound}_{95\%}$ = 600 ms**. (Again, the test was run several times and this result is the worst case.)

Finally we decided to run the same test again, but with the measuring instance being of the same type as the disturbing instance: all optimisations activated and a block size of 1 kB. This measuring instance can be launched with the following command:

```
$ ./hdd -b 1024 -f <file>
```

We discovered that in order to reach a **good result**, the limitation has to be set to 30 MB/s (20 % of 150 MB/s). We compared this result with the one obtained without virtualization (running the measuring instance and the 24 disturbing instances on the same hardware platform):

	Over ESXi	Without virtualization
bound$_{95\%}$	1.0 s	3.0 s
Throughput	30 MB/s	6 MB/s

Table 3-23: Final disk latency test

So we had reached an **acceptable isolation between the virtual machines and a low enough disk access latency.** However we had to sacrifice the maximum disk throughput of each virtual machine. The best practice is to limit the maximum throughput for all virtual machines so that *throughput limitation x number of virtual machines = 80 % x mean throughput of ESXi*.

Concretely in our configuration (RAID 5 over four hard disks), the mean throughput reached by ESXi is 150 MB/s. Which means that:

- with two virtual machines, the limitation has to be 60 MB/s (40 %);
- with three virtual machines, the limitation has to be 40 MB/s (26.7 %);
- with four virtual machines, the limitation has to be 30 MB/s (20 %).

We didn't consider the case with more than four virtual machines because it isn't relevant for this project (see chapter 3 Design) but we can assume that the previous result scales well with a few more virtual machines.

Note: Sacrificing the maximum throughput is not a problem in a real-time system, because it doesn't change the worst cases throughput, which is the throughput when all virtual machines want to access the hard disk drive. With four virtual machines, this worst case throughput is 30 MB/s. If 30 MB/s are enough for each machine, then there is no reason to bewail the loss of the maximum throughput (150 MB/s). It's actually an advantage: as explained in chapter 2.1.3.1 Hard Real Time, reducing the gap between the worst case and the best case improves the predictability of the system and makes it easier to test and dimension.

3.5.1.3.6 Throughput and latency when reading

Until now we only focused on the throughput and the latency when writing into a file. We can assume that the results obtained for writing are also valid for reading, but we need to check that the throughput when reading is greater than or equal to the throughput when writing, otherwise the throughput limitation (which applies for writing and for reading) must be even more restrictive.

The test program developed for this test is similar to the previous one. Instead of writing into a file, the program creates a file with a size of 1 GB and fills it. As soon as the file is filled, the test begins: the program reads blocks sequentially from this file. The main available options are:

- -k: disables kernel buffering (no kernel-space buffer will be used);
- -u: disables user-level buffering (no user-space buffer will be used);
- -b: specifies the size of the blocks (in bytes) to be read from the file on each iteration.

Each read block is involved in a (useless) computation with the same technique as described in chapter 3.5.1.2 Main Memory, in order to be sure that the data are really read from the hard disk drive and that the compiler doesn't "notice" that there is no need to read the data. The full implementation is available on the CD-ROM: `/eval/hdd/read/` .

Without virtualization

Without virtualization, we reach the maximum throughput with a block size greater than or equal to 2 kB and with all optimizations enabled. This throughput is **215.8 MB/s** and the standard deviation is 1 MB/s. No latency was detected. You can run the test yourself with the following command:

```
$ ./hdd -b 2048 -f <file>
```

Over ESXi

Over ESXi, this throughput is **191.7 MB/s** (89 %) and the standard deviation is still 1 MB/s. No latency was detected.

ESXi shows **near-native performance when reading** from the hard-disk drive. Moreover the throughput for reading is higher, which means that the throughput limitation determined in chapter 3.5.1.3.5 Improvement attempt is restrictive enough.

Appendix E Hard-disk latency reduction with ESXi presents a few other solutions explored in order to reduce the hard-disk latency.

3.5.1.3.7 Conclusion

ESXi solves Comsoft's main problem concerning the latency when several applications are accessing the hard-disk drive. To achieve this the *LSI Logic SAS* virtual SCSI controller has to be used and the advanced parameter *Disk.BandwidthCap* has to be set.

3.5.1.4 Network

The test program designed to test the performances when communicating over the network is very similar to the one designed to test the performances when accessing the hard disk drive (see chapter 3.5.1.3 Hard Disk Drive): instead of reading from or writing into a file, a *client* sends TCP packets over the network and a *server* receives those packets.

The full implementation of the client and the server is available on the CD-ROM: `/eval/network/`. You can run all tests yourself with the following commands:

```
$ ./server -b <size of blocks to be read> -p <local TCP port>
$ ./client -b <size of blocks to be sent> -i <IP of the server>
     -p <remote TCP port>
```

3.5.1.4.1 Without virtualization

Two hardware platforms were connected with one Ethernet crossover cable. First we determined the maximal throughput, by running one client on the first platform and one server on the second platform. We reached a net TCP throughput of 114.5 MB/s (915.7 Mb/s) when:

- the block size for the client application was greater than or equal to 128 bytes;
- the block size for the server application was greater than or equal to 512 bytes.

No latency was detected and the standard deviation of the mean throughput is negligible.

In this section, we want to consider not only the throughput and the latency but also the CPU usage (visible with the `htop` command). As a reference, we measured:

- on the client side, a CPU usage of between 7 and 10 % of one execution unit when the block size is 4 kB (about 1 % of the whole CPU);
- on the server side, a CPU usage of between 65 and 80 % of one execution unit when the block size is 4 kB (about 10 % of the whole CPU).

Then we tested what happens when N clients on one side send data to N servers on the other side. Each client only communicates with one server at the same time, thus defining a client/server pair. Data flows are separated by means of different TCP ports. Here are the results:

Number of instances on each side	Net throughput	$\text{bound}_{95\%}$
2	57.5 MB/s (no deviation detected)	No latency detected
4	28.8 MB/s (σ_X = 6.1 MB/s)	50 ms
8	14.4 MB/s (σ_X = 6.6 MB/s)	150 ms

Table 3-24: Several applications using the same network interface

As we can see, the network throughput assigned by Linux to each application is only predictable if less than two applications are using the same network interface. Nevertheless the latency stays reasonable.

3.5.1.4.2 Over ESXi

VMware has implemented in ESXi the concept of *virtual switch*. A virtual switch connects several virtual NICs to several physical NICs, with the following properties:

- If several virtual NICs (typically from several virtual machines) are connected to the same virtual switch, they can exchange Ethernet data just like physical NICs connected to the same physical switch can.
- If several physical NICs are (virtually) connected to the same virtual switch, data coming from virtual NICs and going outside (to the physical network) are distributed over the different physical NICs, either with load sharing or considering one physical NIC as being active and the others as being in standby mode (they will be used only if the active one fails).

To create a virtual switch, click on "vSphere Client / Configuration / Network / Add network...".

You can create up to 10 virtual NICs per virtual machine, which is enough for this project (see chapter 3 Design). Several types of virtual NIC are available: [VV27]

- *Flexible* (default choice): it's an emulated version of the AMD 79C970 PCnet32 LANCE NIC, compatible with most operating systems. However if you installed the VMware Tools (guest additions for device paravirtualization) [VV23], the virtual NIC is used as a *VMXNET* adapter (the VMware adapter making use of paravirtualization) offering better performances, as soon as the VMware Tools are loaded. Therefore this adapter should deliver high performances and is also recognized by the (virtual) BIOS, allowing network boot.

- *E1000*: it's an emulated version of the Intel 82545EM Gigabit Ethernet NIC, compatible with most newer operating systems. VMware Tools are not needed and this adapter allows network boot.
- *VMXNET 2 (enhanced)*: it's an improved version of the VMXNET adapter, supporting features such as jumbo frames and hardware offloads (see chapter 3.4.1 VMware Infrastructure). VMware Tools are needed and therefore network boot is not supported.
- *VMXNET 3*: it's the latest generation of VMware devices using paravirtualization. It supports all features already offered by VMXNET 2 and adds several new features. VMware Tools are needed and therefore network boot is not supported.

Note: The network boot feature (PXE) is not needed by Comsoft products for the moment.

We evaluate in this chapter the properties of each type of virtual network adapter.

Performance

In this part we measured the performance of each network adapter type, separately for the *client* application and for the *server* application. To do so we created one virtual machine on one hardware platform and connected it to another platform not using virtualization:

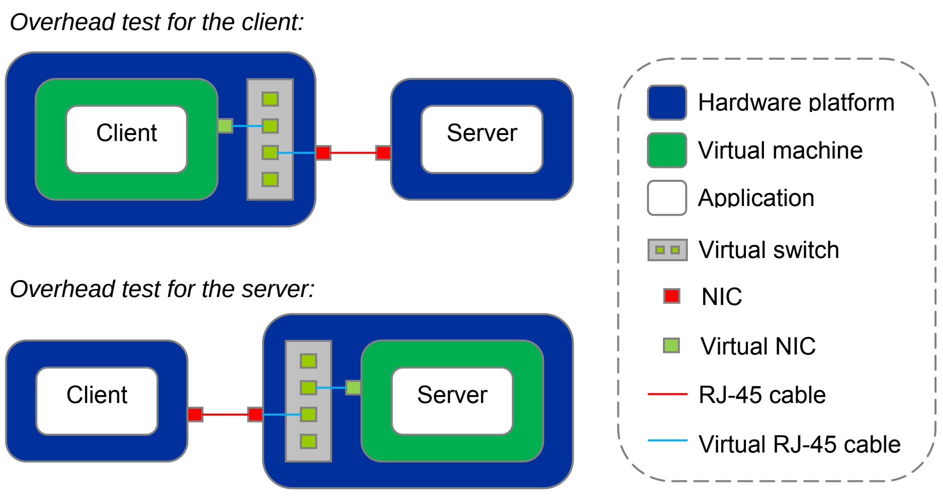


Figure 3-38: Structure of the overhead tests

The application running without virtualization (the server when testing the client, the client when testing the server) uses a block size of 4 kB to make sure it isn't a bottleneck. The following table shows the most relevant results:

Virtual adapter type	Tested application	Block size	Net throughput
Flexible without VMware Tools	Server	512 bytes	17.8 MB/s
		≥ 64 kB	58.4 MB/s
	Client	128 bytes	5.6 MB/s
		≥ 16 kB	13.9 MB/s

Virtual adapter type	Tested application	Block size	Net throughput
Flexible with VMware Tools	Server	≥ 512 bytes	113.1 MB/s (σ_X = 1.1 MB/s)
	Client	≥ 128 bytes	112.3 MB/s (σ_X = 1.0 MB/s)
E1000	Server	≥ 512 bytes	105.4 MB/s (σ_X = 6.7 MB/s)
	Client	128 bytes	92.8 MB/s
		≥ 256 bytes	111.2 MB/s (σ_X = 4.0 MB/s)
VMXNET 2 (enhanced)	Server	≥ 512 bytes	109.9 MB/s (σ_X = 0.8 MB/s)
	Client	128 bytes	106.4 MB/s
		≥ 256 bytes	113.4 MB/s (σ_X = 1.2 MB/s)
VMXNET 3	Server	≥ 512 bytes	111.0 MB/s (σ_X = 0.7 MB/s)
	Client	128 bytes	102.5 MB/s
		≥ 256 bytes	113.9 MB/s (σ_X = 0.7 MB/s)

Table 3-25: Network performance of ESXi

Note: In all cases, **no latency** was detected.

At this point, we can **exclude the Flexible adapter when used without VMware Tools**, because of its poor performance. All **other adapters** showed **near-native performance** with a **good predictability**.

CPU Usage

This test was performed like the previous one, but we measured the CPU usage during the data transfer. To do so, we installed *vSphere Management Assistant*, a special virtual machine provided by VMware for free. [VV27] This virtual machine provides a tool called `esxtop`, similar to `top`, displaying the resource utilization *on the host* (see the manual of `esxtop` and the document *Using esxtop* [VV20]). The CPU usage is visible on the line containing "PCPU UTIL". A block size of 4 kB was used. The following table shows the results:

Virtual adapter type	Tested application	CPU usage (% of the whole CPU)
Flexible with VMware Tools	Client	≈ 12 %
	Server	≈ 12 %
E1000	Client	≈ 14 %
	Server	≈ 17 %
VMXNET 2 (enhanced)	Client	≈ 12 %
	Server	≈ 18 %
VMXNET 3	Client	≈ 12 %
	Server	≈ 18 %

Table 3-26: CPU usage for networking over ESXi

There is no significant difference between the different adapter types concerning the CPU usage. As already shown, the CPU usage of the client application without virtualization was only 1 % and the CPU usage of the server application without virtualization was 10 %. Of course the CPU usage with virtualization is higher but we are not able to say at this point if it's acceptable or not: life-size tests with Comsoft products have to be made.

Isolation

The goal of this test is to determine how ESXi assigns the network throughput to several virtual machines using the same physical NIC. For example the following figure shows the structure of the test when testing the isolation of three clients using the same physical interface. On the side without virtualization, three different NICs are used to make sure that this side is not a bottleneck. All instances of the test program used a block size of 4 kB.

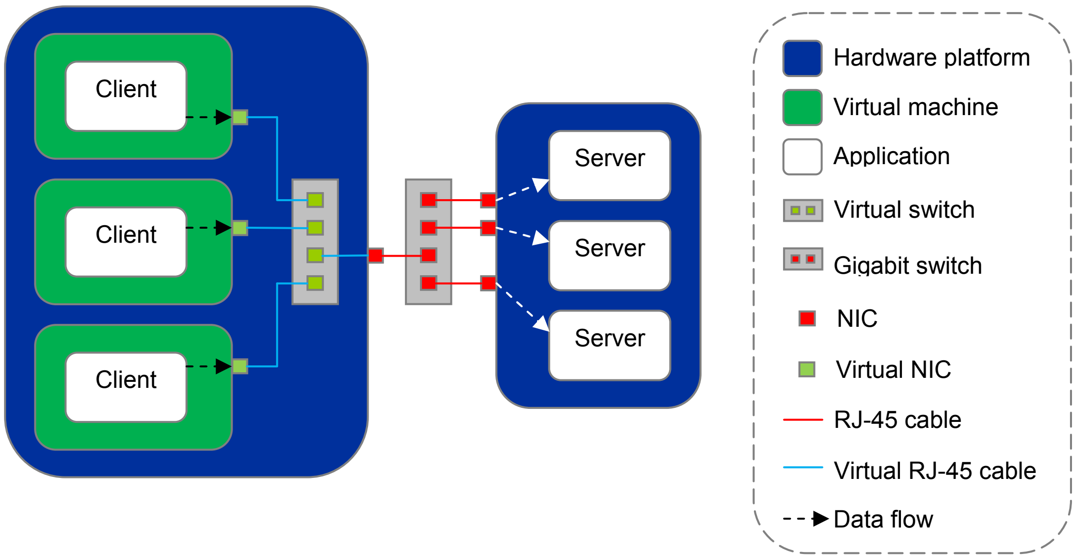


Figure 3-39: Structure of the isolation test between three clients

The following table shows the most relevant results:

Virtual adapter type	Tested isolation	Net throughput	$bound_{95\%}$
Flexible with VMware Tools	2 clients	57.4 MB/s (σ_X = 21.6 MB/s)	No latency detected
E1000	2 clients	58.1 MB/s (σ_X = 0.9 MB/s)	50 ms
	2 servers	57.4 MB/s (σ_X = 2.1 MB/s)	No latency detected
	3 clients	38.3 MB/s (σ_X = 0.7 MB/s)	No latency detected
	3 servers	38.3 MB/s (σ_X = 9.7 MB/s)	50 ms
	6 clients	19.9 MB/s (σ_X = 2.7 MB/s)	100 ms
	6 servers	19.2 MB/s (σ_X = 7.7 MB/s)	100 ms
VMXNET 2 (enhanced)	2 clients	57.4 MB/s (σ_X = 1.1 MB/s)	50 ms
	2 servers	57.4 MB/s (σ_X = 1.7 MB/s)	No latency detected
	3 clients	38.3 MB/s (σ_X = 12.8 MB/s)	1.6 s
	3 servers	38.3 MB/s (σ_X = 8.9 MB/s)	No latency detected

Virtual adapter type	Tested isolation	Net throughput	$\text{bound}_{95\%}$
VMXNET 3	2 clients	113.1 MB/s (σ_X = 1.9 MB/s)	50 ms
	2 servers	113.1 MB/s (σ_X = 2.3 MB/s)	No latency detected
	3 clients	113.1 MB/s (σ_X = 15.5 MB/s)	50 ms
	3 servers	113.1 MB/s (σ_X = 0.4 MB/s)	50 ms
	6 clients	113.1 MB/s (σ_X = 13.6 MB/s)	1.2 s
	6 servers	113.1 MB/s (σ_X = 8.8 MB/s)	100 ms

Table 3-27: Networking isolation with ESXi

As we can see, **E1000 offers the best results**. The throughput with Flexible has a big deviation, which means a bad isolation among virtual machines. The latency and the standard deviation of the throughput with VMXNET 2 and 3 is higher than with E1000.

Moreover the deviation with E1000 is sometimes also not negligible but as we can see, it only happens when the virtual machines are receiving TCP packets, not when they are sending. It also only happens when more than two virtual machines are connected to the same physical NIC. **In all cases the latency stays very low**.

If this leads to problems when life-size tests with Comsoft products are made, they can be solved by one of the following means:

- The solution has to be designed so that at most two virtual machines are connected to one physical NIC: it may of course increase the number of NICs needed.
- Traffic shaping has to be configured either in the guest operating systems or on the physical switches.

Note: ESXi gives the possibility to set up some traffic shaping, but it only applies to *outgoing traffic*. [VV7] [VV28] No advanced option similar to Disk.BandwidthCap (see chapter 3.5.1.3.5 Improvement attempt) for the network bandwidth was found.

Communication over the virtual network

We tested the communication between two virtual machines of the same hardware platform. The following figure shows the structure of this test:

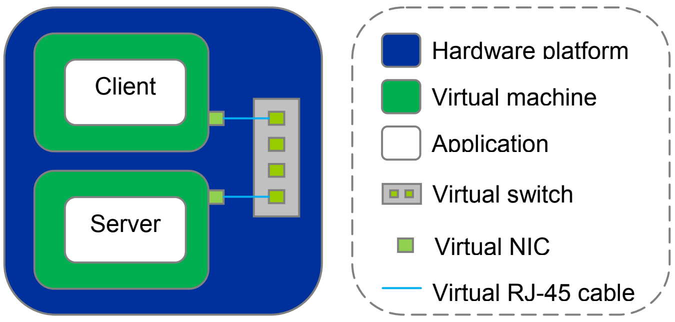


Figure 3-40: Test of the communication over the virtual network

A block size of 4 kB has been used on both sides. This table shows the results:

Virtual adapter type	Net throughput	$bound_{95\%}$	CPU usage (% of the whole CPU)
E1000	124.4 MB/s (σX = 1.6 MB/s)	50 ms	21 %
VMXNET 2 (enhanced)	160.3 MB/s (σ_X = 4.8 MB/s)	50 ms	18 %
VMXNET 3	129.8 MB/s (σ_X = 9.7 MB/s)	100 ms	19 %

Table 3-28: Communication over the virtual network

E1000 offers the **best predictability and latency** with a CPU usage equivalent to the other adapter types.

Even though E1000 has the lowest throughput, this **throughput** is still **better than the native one** (throughput while communicating over a physical network). This improvement was expected (see chapter 2.1.2.1 Reasons).

3.5.1.4.3 Conclusion

ESXi guarantees a **short latency** to guest operating systems using network adapters. It provides **a good throughput isolation with a high predictability**, except when three or more virtual machines receive TCP packets from the same overloaded physical NIC. In case this turns out to be a problem during the life-sized tests, solutions have been proposed.

To reach those results, the **E1000** virtual NIC **has to be used**.

3.5.1.5 Human-Computer Interface

Of course screen, keyboard and mouse have to be virtualized too (into *virtual screen*, *virtual keyboard* and *virtual mouse*), in order to provide each virtual machine with a human-computer interface. This is **usually** implemented as follows:

- the host's graphical interface provides the possibility to open a *window* for each virtual machine, representing its *virtual screen*;
- when one of these windows is *focused*, all actions on the physical mouse or on the physical keyboard are transmitted to the corresponding virtual machine, as if these actions were done on the *virtual mouse* or the *virtual keyboard*.

This is how KVM and VirtualBox work, for example, but it isn't exactly the way ESXi works: ESXi is designed to be as thin as possible. Therefore the interface displayed on the screen of the host is just a *semi-graphical interface* (a.k.a. *color text mode*) and the mouse is not supported. With ESXi, the virtual machines' interfaces are not displayed on the host but on the *management computer* running *vSphere Client* over Microsoft Windows.

The human-computer interface is subject to the "real-time" requirement 2.3.1.1.4.(10), limiting its reaction time. However this requirement is reasonably not to be understood as needing a detailed proof but rather as an indication that the interface must react pretty quickly and never freeze or give the impression of freezing.

vSphere Client

The vSphere Client interface works fine but has several disadvantages. First of all it can be slow: when moving the mouse and/or a window in the guest operating system, you might notice a refresh frequency of less than 15 images per second, which can be very uncomfortable when working on the system during several hours. The interface may seem to *lag behind*, giving the overall impression that the system is slow. The **problem** is drastically **reduced** if you connect the *management computer* directly or through a **gigabit** network to the host.

When the interface of one virtual machine hasn't been used during several days, it can be extremely slow when used again. **This problem disappears after about 20 seconds of use** and is probably due to some long-term scheduling policy concerning the graphical interface.

Another disadvantage is that when a virtual machine is overloaded, the graphical interface (which probably has a very low priority) gets even slower. The installation of VMware Tools (guest additions for device paravirtualization) [VV23] doesn't reduce the problem.

The last disadvantage is that vSphere Client is a Windows application. We can imagine that a solution to automate the installation of ESXi could be developed by Comsoft by means of the different APIs provided by VMware. vSphere Client wouldn't be needed to configure the virtualization layer anymore, except for the human-computer interface of each virtual machine. **It would then be preferable to find another solution running on Linux**, so that Comsoft doesn't have to deliver any Windows workstations. The following parts present a few Linux solutions.

X11-Forwarding

X11-Forwarding corresponds to the use of "`ssh -X`" or "`ssh -Y`" (see the manual of the `ssh` command) under Linux to establish a connection to one guest from a Linux workstation. This technique is **very effective**, even if the workstation is not connected to the server over a "fast" connection (crossover or gigabit). Two problems remain:

- This solution doesn't allow viewing of the guest's desktop (XFCE desktop).
- This solution only works once the guest operating system is installed and configured.

VNC on the guest

Virtual Network Computing (VNC) is a well known platform-independent system to remotely control another computer. It transfers the keyboard events, the mouse events and the screen changes over the network. A *VNC server* has to be started on the guest operating system (see the manual of `vncserver` and `vncviewer` under Linux). *NX* is also a very powerful alternative (only for a remote access to an *X11 server*).

This solution allows the visualization of the guest's desktop but, again, only works once the guest operating system is installed and configured.

VNC on the host

This is how the vSphere Client display works: it uses a VNC connection to the host. It is possible to visualize each guest from another VNC client (e.g. TinyVNC under Linux). To enable the VNC access to the virtual machine "VM1":

- Shut down the virtual machine.

- You need to edit the file `VM1.vmx` . One possibility to do so:
  - Do a right click on the datastore in vSphere Client / Host Configuration / Storage, and choose "Explore datastore...". Find the file and download it.
  - Append the following lines at the end of the file:

```
remotedisplay.vnc.port="5900"
remotedisplay.vnc.enabled="true"
remotedisplay.vnc.password = "password"
```

  - Upload the file into the datastore.
- Start the virtual machine.

You can now access the virtual machine from a Linux workstation with the following command:

```
$ vncviewer <IP_of_the_host>:5900
```

You will have to choose a different port for each virtual machine.

This method allows to monitor a virtual machine, even before it boots any operating system, which means that the virtual BIOS for example is accessible with this method. Surprisingly it does not suffer from the performance disadvantages of vSphere Client and the display is quite flowing.

Conclusion

Different methods exist to control and monitor a guest. Some are less efficient but don't require the guest operating system to be started, whereas some others are more efficient and convenient. **No Windows workstation is required to monitor and act inside a virtual machine, since Linux alternatives exist.**

3.5.1.6 Conclusion

ESXi showed a **good isolation** between virtual machines, a **high predictability** and **low latencies** in comparison to the time granularity of Comsoft products (see chapter 2.1.1 Comsoft products). Therefore ESXi can be a **solution to the problem** exposed by Comsoft. Questions concerning the *sizing* of the system still remain: Are 12 GB RAM enough? Is the now lower hard-disk throughput enough? etc. Those questions can only be answered by *life-sized tests* (see chapter 5 Verification and Validation).

All tests exposed in this chapter (3.5.1 VMware ESXi) can constitute a **reference for testing any other virtualization product** or even an operating system.

3.5.2 KVM

When testing KVM, **Hyperthreading was activated** in the BIOS of both platforms, in contrary to most of the CPU tests performed on ESXi. Therefore each hardware platform has 16 execution units.

The tests presented in chapter 3.5.1 VMware ESXi have of course been used to evaluate KVM too. The document *Getting started with virtualization* [VK1] gives some details on the installation of KVM on Fedora.

3.5.2.1 CPU Utilization

3.5.2.1.1 Overhead on CPU-Bounded applications

This test aims to measure the *approximate* overhead when running a *CPU-bounded* [C8] program (with little or no memory and I/O access) over the hypervisor. We compared the execution time of a CPU-bounded program [C8] on following systems:

- A system with simply a standard installation of Fedora Core 11 32-bits (i386) [C17], without virtualization. This Linux distribution is the most recent one delivered by Comsoft to its customers.
- A system with KVM 0.12.2 [VK3] over a standard installation of *Fedora Core 11 64-bits* (a 64-bits host kernel is needed for a complete use of KVM). Here are the relevant parameters for this test:
  - 10 identical virtual machines are created, with following parameters:
    - OS type: Linux;
    - Version: Fedora 11;
    - CPUs: 16 (maximum value), this means that the guest operating system will have the feeling that it's working with 16 *execution units* and will be able to run 16 threads in parallel.
    - Virt Type: kvm.
    - Architecture: x86_64.
  - A standard installation of Fedora Core 11 32-bits is performed on each virtual machine.

Part 1

The test was run with M = 10 000 (number of iterations for each thread), different numbers of threads (T) and 5 times for each value of T. The structure of the test program can be found in chapter 3.5.1.1.1 Overhead on CPU-Bound applications. The following figure shows the results:

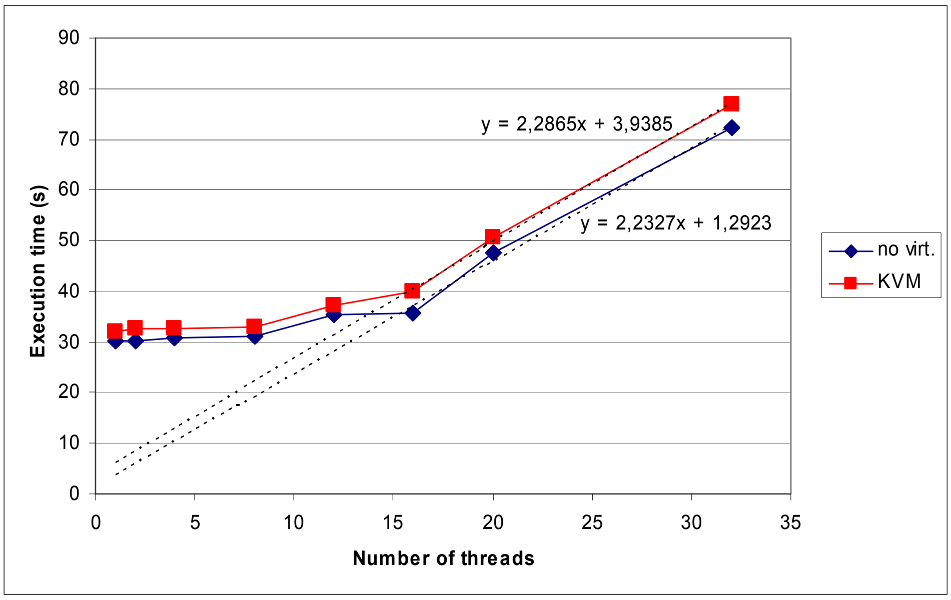


Figure 3-41: Overhead on CPU-Bounded applications (1 VM)

These results are similar to those obtained with ESXi, except for the following differences:

- The virtual machine is able to use the 16 available execution units. However the curves aren't perfectly horizontal for T ≤ 16.
- **The maximum overhead caused by KVM on a long execution of a CPU-bounded program in one virtual machine is about 12 %** (5 % with ESXi) and appears when 16 (Linux) threads have to be scheduled.

The *standard deviation* didn't seem to be increased because of KVM. **KVM doesn't add any significant indeterminism on a long execution of a CPU-bounded program in one virtual machine.**

Part 2

In the second part of this test, the test program is run simultaneously in 3 virtual machines, with T = 8 and M = 10 000. This makes a total of 24 threads. The structure of the test program can be found in Part 2 of chapter 3.5.1.1.1 Overhead on CPU-Bound applications

The expected execution time (without overhead in comparison to the previous results with virtualization) is: (see figure 3-41)

2.2865 * 24 + 3.9385 = **63.9 s**

The results are a mean execution time of 67.6 s and a standard deviation of 2.4 s. They show that:

- The mean **overhead** induced by KVM on a long execution of CPU-bounded programs in 3 virtual machines is **5.8 %** (in comparison to the result with only one virtual machine), which is low (even though there was no overhead with ESXi on 8 execution units).
- The *standard deviation* is slightly increased because of KVM, but only represents 3.6 % of the whole execution time, which is acceptable. (ESXi didn't increase the indeterminism at this stage.)

Part 3

In the third step, the same test is performed with 10 virtual machines. The expected execution time (without overhead in comparison to the execution time with only one virtual machine) is: (see figure 3-41)

2.2865 * (10 * 8) + 3.9385 = **186.9 s**

Here are the results:

		Execution time (± 0.5 s) / VM N°										Per test (s)	
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Mean**	**σ_X**
Test N°	**1**	235.5	233.5	237.5	237.5	234.5	225.5	235.5	237.5	237.5	236.5	235.1	3.7
	2	260.5	263.5	260.5	259.5	252.5	253.5	260.5	257.5	262.5	266.5	259.7	4.3
	3	272.5	266.5	273.5	279.5	270.5	265.5	270.5	265.5	271.5	267.5	270.3	4.3
	4	256.5	258.5	253.5	262.5	252.5	257.5	249.5	261.5	256.5	262.5	257.1	4.4
	5	239.5	245.5	243.5	242.5	230.5	231.5	234.5	243.5	244.5	247.5	240.3	6.1
Per VM (s)	**Mean**	252.9	253.5	253.7	256.3	248.1	256.7	250.1	253.1	254.5	256.1	Result: 252.5 s (σ_X = 13.8 s) ➔ 35 % overhead	
	σ_X	15.3	13.8	14.2	16.8	16.1	17.3	15.7	12.0	13.7	13.6		

Table 3-29: Overhead on CPU-Bounded applications (10 VMs)

Those results show that:

- The mean **overhead** induced by KVM on a long execution of CPU-bounded programs in 10 virtual machines is **35 %** (in comparison to the execution time without virtualization), which is **high**. (ESXi didn't induce any overhead with 8 execution units.)
- **The *standard deviation* per virtual machine is higher** but still represents only 5.5 % of the total execution time. It's **acceptable**.

KVM doesn't scale well with the number of virtual machines, but we only plan to create three virtual machines per host in this project (see chapter 1 Introduction).

The results of the three parts of this test are **satisfactory**, even if the results are not as good as with ESXi.

3.5.2.1.2 Overhead on system calls

This test has been described in chapter 3.5.1.1.2 Overhead on system calls. The test was run with K = 100 000, different numbers of threads (T) and 5 times for each value of T. The following figure shows the results. The measured standard deviation was the same on both platforms (max. 2.4 s).

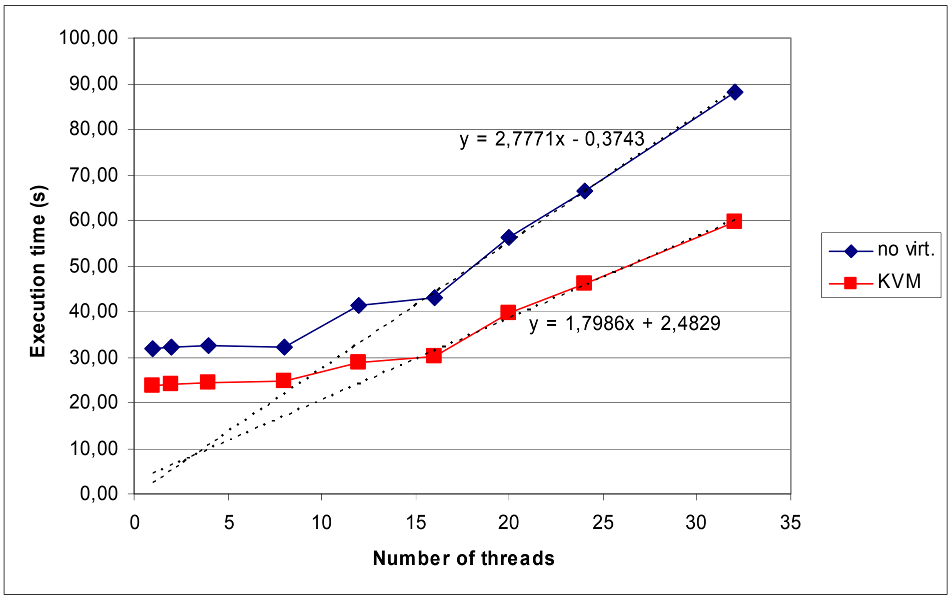


Figure 3-42: Overhead on system calls

Those results show that:

KVM brings an **improvement of about 26 %** on this microbenchmark running as many system calls as possible. (We observed an overhead of 9 % with ESXi using 8 execution units. However a similar improvement has been observed using 16 execution units.)

- **KVM doesn't add any significant indeterminism on a long execution of this microbenchmark.**

The results of this test are more than **satisfactory**, because of the significant improvement observed.

3.5.2.1.3 MMU Performance

This test has been described in chapter 3.5.1.1.3 MMU Performance. It was run with K = 10 (number of iterations per thread). Here are the results for KVM:

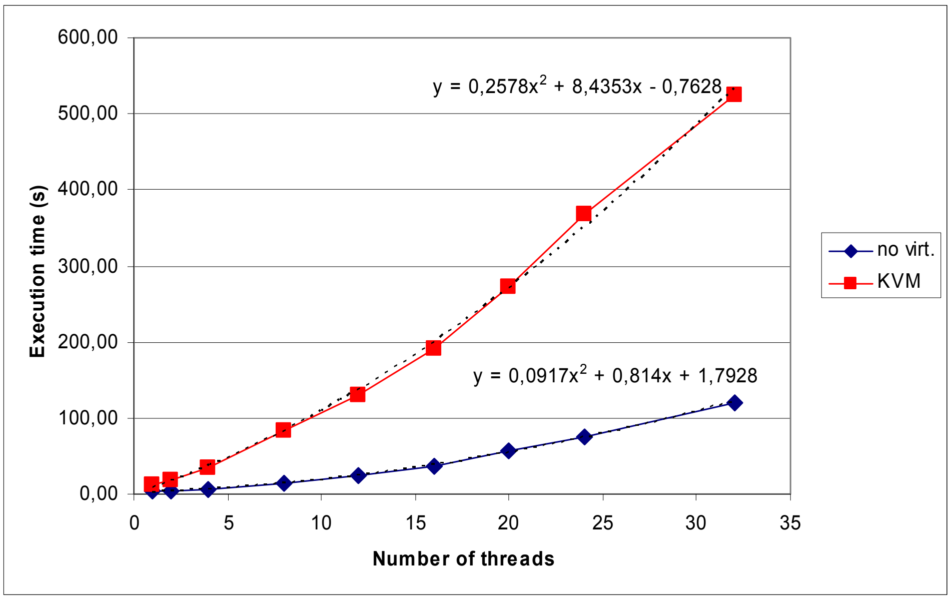


Figure 3-43: Overhead on MMU usage

Those results show a **huge overhead** (between 230 % and 540 %). (The observed overhead with ESXi over 8 execution units was 38 %.) We tried to reduce it by making the following changes:

- With "Architecture = i686", the results stay unchanged.
- With "Virt Type = qemu" and "Architecture = x86_64", following problems appear:
  - If the RAM allocated to one virtual machine is greater than 4079 MB, it can't start and the error notification "could not query memory balloon allocation" is displayed.
  - The virtual machine often freezes and has to be rebooted. It seems unstable, even when ACPI and/or APIC in the guest is disabled (by means of options passed to the guest kernel in the GRUB menu).
  - The virtual machine seems extremely slow.
- With "Virt Type = qemu" and "Architecture = i686", following problems appear:
  - The guest Linux freezes when trying to detect the available execution units (at the very beginning of the boot process), even with ACPI and/or APIC disabled in the guest.
  - Before freezing, the virtual machine seems extremely slow.

Even **the standard deviation is high**: with two threads, it represents **more than 12 %** of the whole execution time. However we decided to continue testing KVM.

3.5.2.1.4 CPU time distribution

The goals and details of this series of tests are explained in chapter 3.5.1.1.4 CPU time distribution.

Part 1

First we ran the test presented in Part 1 of chapter 3.5.1.1.4 CPU time distribution, with M = 10 000 (number of iterations per thread), T_A = 8 (number of threads in the first virtual machine) and T_B = 16 (second virtual machine). This time 16 execution units are available.

With similar calculations, we can deduce that:

- if KVM distributes the CPU time equally among threads, $u_A = u_B$ is to be expected (see figure 3-44);
- if KVM distributes the CPU time among virtual machines equally or proportionally (to their needs), $u_B = 1.5 \times u_A$ is to be expected (see figure 3-45).

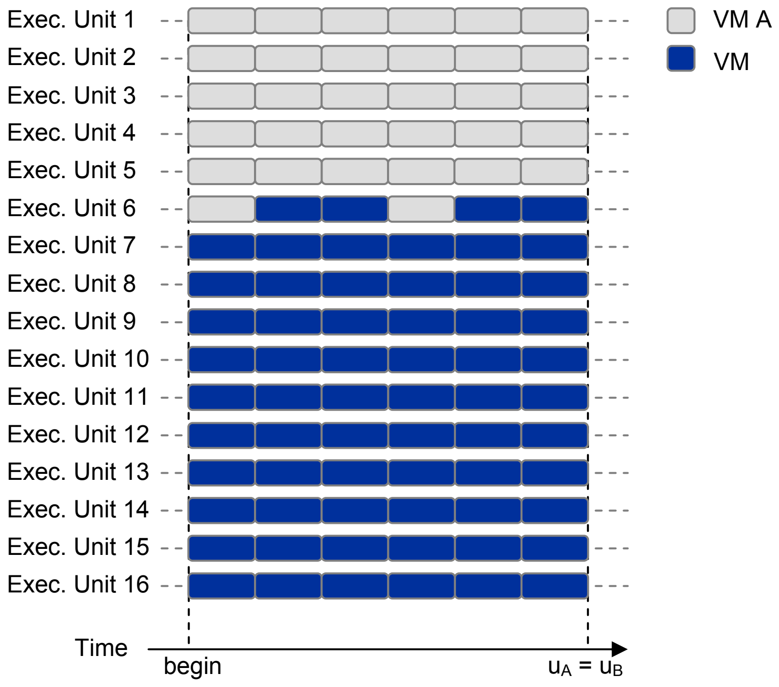


Figure 3-44: Equal CPU time distribution among threads

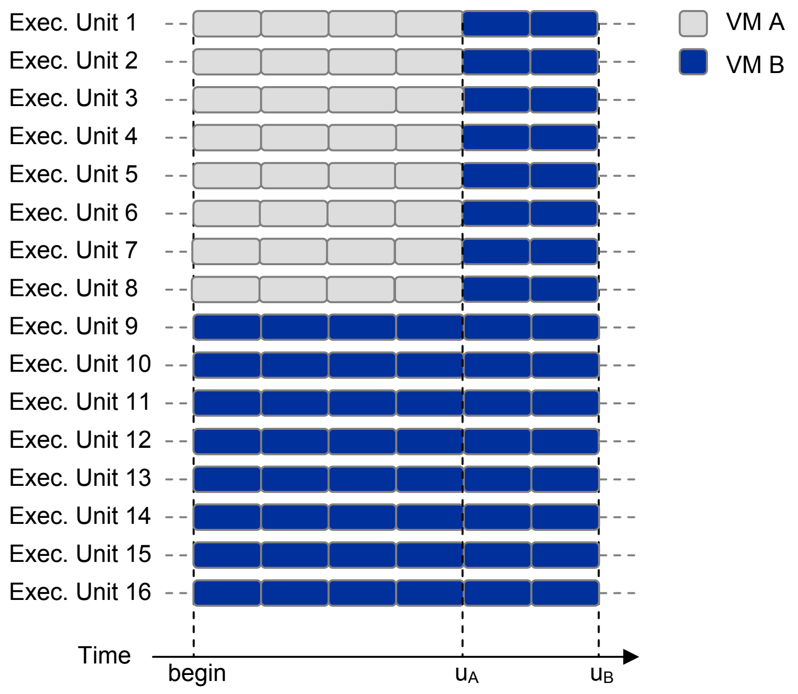


Figure 3-45: Equal or proportional CPU time distribution among virtual machines

The result of this test is $u_B = 1.39 \times u_A$, which doesn't allow us to determine the used scheduling strategy. This is probably due to the overhead induced by KVM, which distorts the result.

Part 2

We decided to draw the curves defined in Part 3 of chapter 3.5.1.1.4 CPU time distribution, to visually identify the scheduling strategy of KVM. Figure 3-46 shows the result:

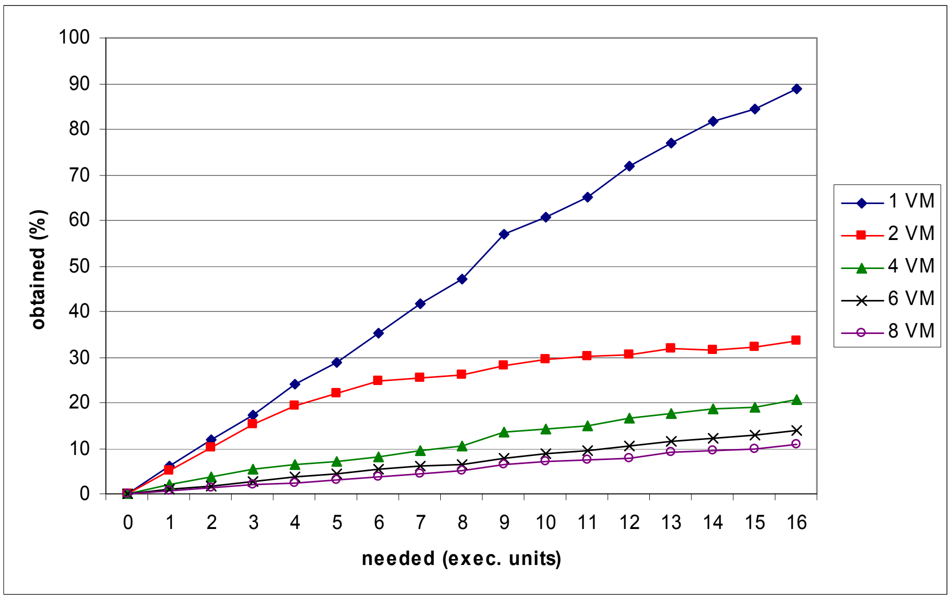


Figure 3-46: KVM's CPU time distribution among virtual machines

The CPU time distribution done by KVM looks like a distribution among virtual machines proportionally to their needs (see figure 3-16) distorted by a non-negligible overhead. As explained, this CPU time distribution delivers **few guarantees** and a **bad isolation** between virtual machines.

3.5.2.2 Conclusion

We decided to stop the evaluation of KVM at this point. KVM may be a good virtualization product but it **isn't** currently **adapted to our project**. It added **significant indeterminism** on some microbenchmarks and provides a **poor isolation** between virtual machines concerning the CPU distribution.

4 Implementation

4.1 Architecture

Figure 4-1 shows the architecture of our implementation.

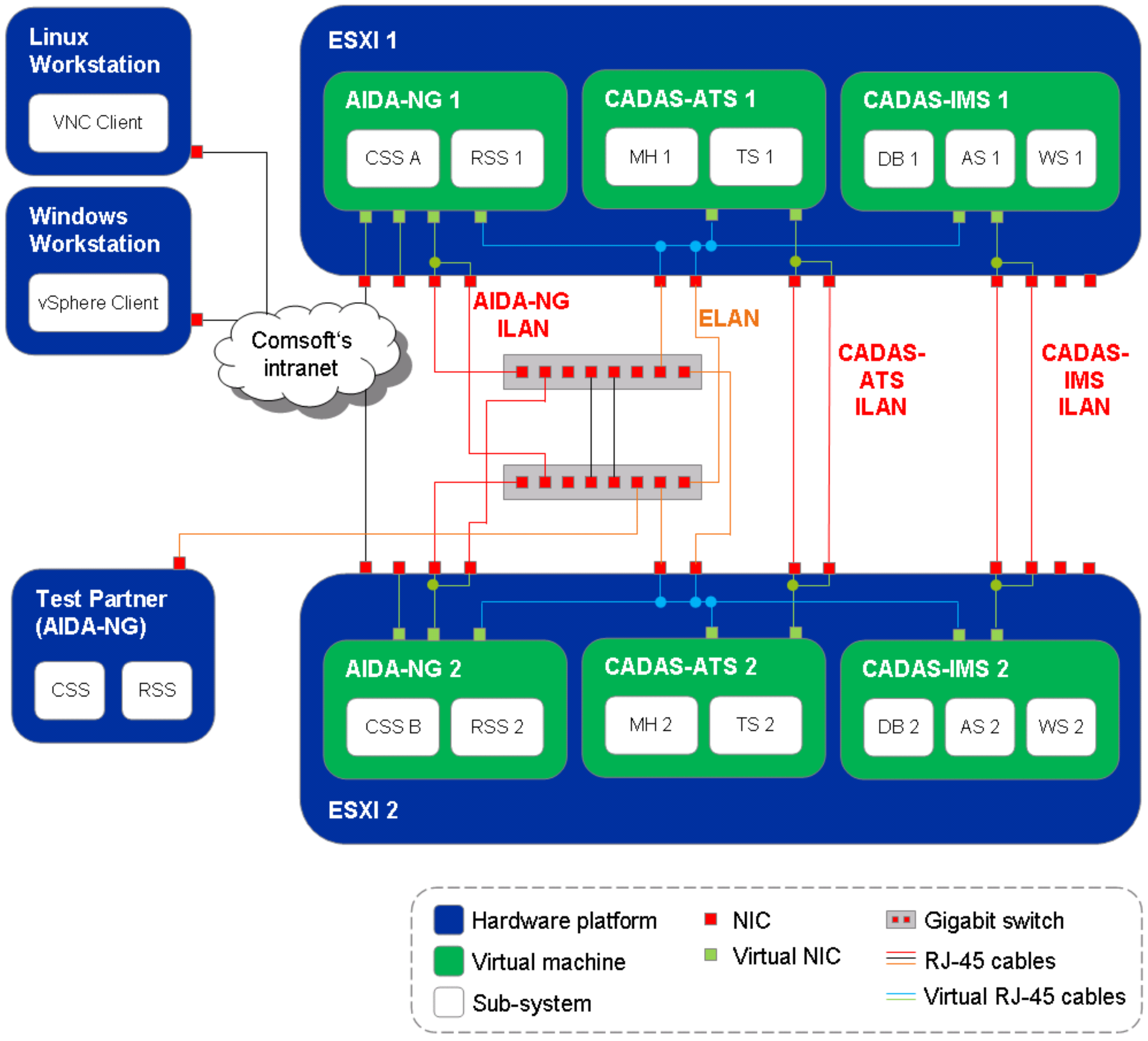


Figure 4-1: Architecture of the implemented solution

Hardware platforms

The solution is based on two servers (HP ProLiant DL380 G6) running a virtualization product (ESXi), as anticipated during the design phase (see chapter 3 Design). At least one external workstation is needed to administrate ESXi and display the virtual machines' human-computer interfaces (see chapter 3.5.1.5 Human-Computer Interface). For testing purposes we decided to use two such *management workstations* ("Linux Workstation" and "Windows Workstation").

In order to test our solution, a *test partner* is needed, representing the solution of another airport/country. Therefore this hardware platform is not part of our solution.

Physical networks

The networks "AIDA-NG ILAN", "CADAS-ATS ILAN", "ELAN" and "CADAS-IMS ILAN" are similar to what was anticipated during the design phase, but "AIDA-NG ILAN" and "ELAN" share the same switches. Comsoft often makes this simplification and separates both networks on level 2 (Ethernet level) with different VLAN-IDs.[1]

The physical NICs used to communicate with other airports/countries (see "WAN" in chapter 3 Design) are not used here. They are normally connected to Comsoft devices named *SNL Boards* (Serial Network Link), converting Ethernet lines to serial lines (e.g. *V.24*) and vice versa.

Our test partner is connected to our solution over the ELAN, instead of using SNL Boards. This is how Comsoft usually tests the performances of its systems.

The management workstations are connected to the servers over the Comsoft's internal network. This allows us to test the speed of the different human-computer interfaces over a slower network.

Virtual networks

The virtualization product used in our solution is VMware ESXi, and the virtual networks are implemented by means of virtual switches. As explained in chapter 3.2 New design, a classical solution uses a *Linux bonding* interface in a *hot-standby mode*, so that if one network link fails, the second one is used, as shown in figure 4-2.

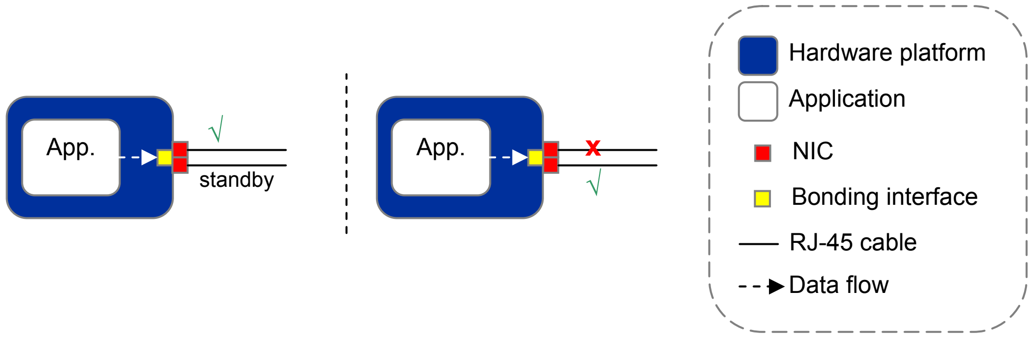


Figure 4-2: Redundancy with Linux bonding

The natural way to implement the same mechanism over a virtualization product would be to let Linux deal with bonding as usual and just map one virtual NIC to one physical NIC. However it can't work with ESXi because of the *virtual switches*: **a virtual link can't fail** (except if an administrator disables it manually in vSphere Client). Therefore the bonding interface can't detect that the physical link failed, as shown in figure 4-3.

[1] The redundant link between the two switches is then called a *redundant trunk link*.

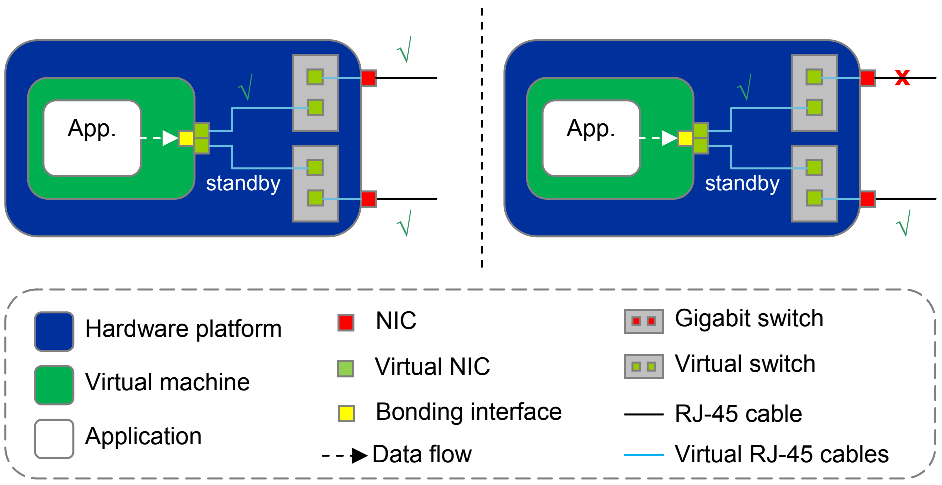


Figure 4-3: Linux bonding and VMware ESXi

The correct solution is to let ESXi do this work (hot-standby) instead of Linux, as shown in figure 4-4 and as explained in chapters 3.2 New design and 3.5.1.4.2 Over ESXi.

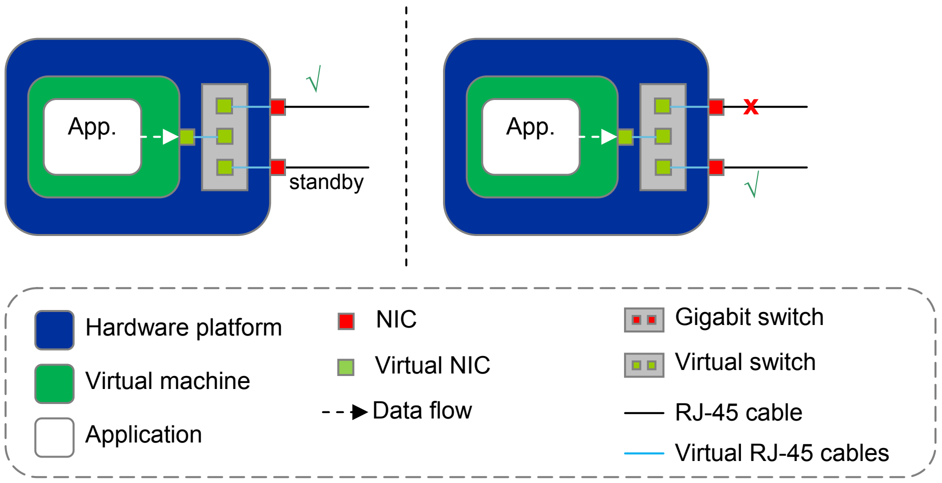


Figure 4-4: ESXi's hot-standby mode

Using the test program presented in chapter 3.5.1.4 Network, we tested the time needed to perform this *line switching*, when the first cable is unplugged while data is being sent over or received from the network. The test program didn't suffer from any TCP disconnection: only a latency/pause of about 600 ms (it varies between 500 and 700 ms) appeared during the exchange when unplugging the first cable. This is **acceptable** for Comsoft.

One virtual machine (AIDA-NG 1) is connected to Comsoft's intranet, so that X11-Forwarding (see chapter 3.5.1.5 Human-Computer Interface) can be tested from the Linux workstation too. From this virtual machine we can access the others over the ELAN and use X11-Forwarding for them too (*cascading X11-Forwarding*).

4.2 Details

Hardware platforms

Except for the points mentioned below, the hardware platforms supporting the hypervisor are as described in chapter 3.5 Evaluation.

Hyperthreading was disabled in the BIOS. A firmware upgrade of the platform was made with HP Firmware Update CD 8.70 (latest available version on 2010-02-04).

We only created one RAID 5 group on three hard-disk drives: A1, A2 and A3 (see figure 3-4). All other drives are unnecessary and could have been removed. Appendix F Hard-disk space needed for the project shows why three hard-disk drives are sufficient for this project.

Physical networks

Both Gigabit switches are *Cisco WS-C3560G-24TS-S*. [N3] [N4] The configuration files of these switches are available on the CD-ROM: `/impl/cisco/` . The procedure for configuring Cisco switches can be found in *Comsoft's LAN Installation Guide*. [A19]

Hypervisor

ESXi 4.0.0 Update 1 Build 208167 (last available version on 2010-03-23) was installed on both hardware platforms and all available patches have been applied with the Host Update Utility. The version of ESXi is then "Build 219382". Its configuration is based on the conclusions of chapter 3.5.1 VMware ESXi, except for the disk bandwidth limitation, as described below.

Based on chapter 3.5.1.3 Hard Disk Drive, we measured the maximum write throughput in one virtual machine on a RAID 5 group over three hard disk drives. The result is about 100 MB/s. Since three virtual machines have to be created on each hardware platform, the parameter `Disk.BandwidthCap` should have been set to 27 306 kB/s.[1] However we observed that the virtual machines didn't run the installation of their guest operating system (this procedure writes heavily on the hard-disk) at the same speed as each other, indicating that the limitation wasn't restrictive enough. We decided to set this limitation to 20 000 kB/s. This fixed the problem. Chapter 5.1 Hard disk latency shows that this setting brings reliable results.

In contrary to chapter 3.5.1 VMware ESXi, a **free license** has been used for this implementation. This free version of ESXi happens to come with a new limitation: a virtual machine can only use 4 execution units (8 are available on each hardware platform). If you try to assign more than 4 execution units to one virtual machine, ESXi refuses to start it.

This limitation is just a commercial limitation. It shouldn't change the results of chapter 3.5.1.1 CPU Utilization concerning the predictability of ESXi. Moreover it doesn't fundamentally reduce the performance of the system: as explained in chapter 2.1.3.1 Hard Real Time, the worst case is the only important indicator in a real time system. Without limitation of ESXi and with three virtual machines, a virtual machine would get, if it needs it, 8 execution units in the best case and 2.67 in the worst case. With a free license, the worst case stays the same but the best case is only 4 execution units. If all three virtual machines need to execute several threads at the same time, the 8 available execution units will be used: there is no waste.

[1] 100 MB/s x 80 % / 3 = 27 306 kB/s

However chapter 3.5.1.1.5 Scheduling demonstrated that reducing the number of virtual CPUs of each virtual machine may lead to a longer latency for a virtual machine to get an execution unit. So if the latency tests fail for this implementation (see chapter 5.1 Hard disk latency), it may mean that the CPU latency is too long and the full version of ESXi will have to be tried.

The virtual machines must be automatically started when ESXi starts, so that this part of the solution easily starts up after a power failure. Following parameters have been set in vSphere Client / Configuration / Software / Virtual Machine Startup-Shutdown:

- Allow virtual machines to start and stop automatically with the system: yes;
- Default Startup Delay: 0 seconds;
- For each virtual machine:
  - Any Order;
  - Startup Delay: 0 seconds.

The complete configuration of ESXi can be found on the CD-ROM: `/impl/esxi/esxi1.txt` and `/impl/esxi/esxi2.txt` .

Virtual machines

Their configuration is based on the conclusions of chapter 3.5.1 VMware ESXi, except for the assigned main memory, as described below.

Chapter 3.5.1.2 Main Memory explains that there is no reason to limit the amount of main memory usable by a virtual machine, because 12 GB are available and all Comsoft products need less than 4 GB.[1] However if each virtual machine is allowed to use 12 GB of main memory, the Linux guest operating system will see a lot of unused main memory and feel free to use it to buffer I/O operations.

One day we noticed that the virtual machine "CADAS-ATS 1" was using 6 GB of main memory. We decided then to **limit each virtual machine to 3.9 GB RAM**. We now have the guarantee that the system will never *swap*, because:

- our guest operating systems have no *swap partition*;
- 3.9 x 3 < 12 GB: the hypervisor has no reason to swap.

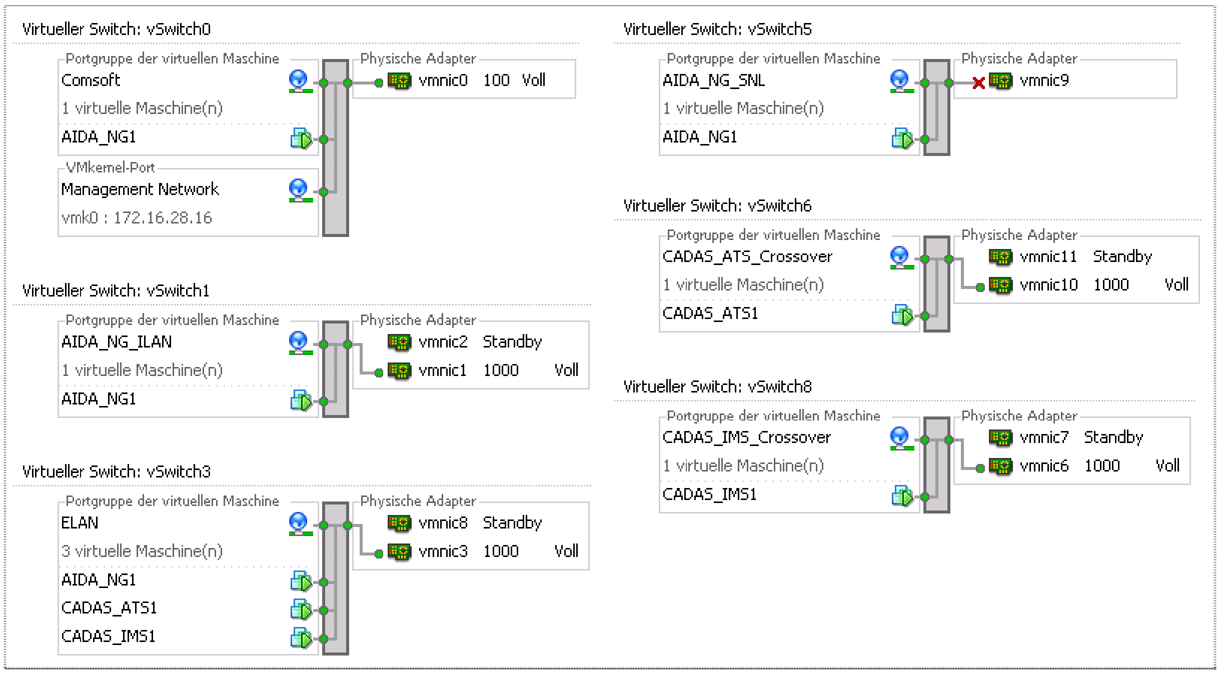
By default, 4 MB *video-RAM* are assigned to each virtual machine. It may not be enough to start the virtual machine. We recommend to set the maximum value: 128 MB. This can be configured in vSphere Client / Virtual machine configuration / Hardware / Graphic card.

The configuration of each virtual machine can be found on the CD-ROM: `/impl/esxi/` .

Virtual networks

Figure 4-5 shows the configuration of the virtual networks on the first hardware platform. The configuration of the second platform is similar.

[1] 4 GB/VM x 3 VM = 12 GB



Virtueller Switch: vSwitch0
Portgruppe der virtuellen Maschine
Comsoft
1 virtuelle Maschine(n)
AIDA_NG1
VMkernel-Port
Management Network
vmk0 : 172.16.28.16
Physische Adapter
vmnic0 100 Voll

Virtueller Switch: vSwitch1
Portgruppe der virtuellen Maschine
AIDA_NG_ILAN
1 virtuelle Maschine(n)
AIDA_NG1
Physische Adapter
vmnic2 Standby
vmnic1 1000 Voll

Virtueller Switch: vSwitch3
Portgruppe der virtuellen Maschine
ELAN
3 virtuelle Maschine(n)
AIDA_NG1
CADAS_ATS1
CADAS_IMS1
Physische Adapter
vmnic8 Standby
vmnic3 1000 Voll

Virtueller Switch: vSwitch5
Portgruppe der virtuellen Maschine
AIDA_NG_SNL
1 virtuelle Maschine(n)
AIDA_NG1
Physische Adapter
vmnic9

Virtueller Switch: vSwitch6
Portgruppe der virtuellen Maschine
CADAS_ATS_Crossover
1 virtuelle Maschine(n)
CADAS_ATS1
Physische Adapter
vmnic11 Standby
vmnic10 1000 Voll

Virtueller Switch: vSwitch8
Portgruppe der virtuellen Maschine
CADAS_IMS_Crossover
1 virtuelle Maschine(n)
CADAS_IMS1
Physische Adapter
vmnic7 Standby
vmnic6 1000 Voll

Figure 4-5: Virtual networks

Guest operating system and CCMS

The guest operating system of each virtual machine is a Fedora 11 32-bits installed by means of the following media:

- *Comsoft's Fedora 11 i386 DVD 2010-01-28*;
- *Comsoft's Kickstart CD* built from CCMS R09.2 V2.0.2816.

The version R09.2 V2.0.2816 of CCMS has been installed (see chapter 2.1.1 Comsoft products). In order to be able to start the X server, following platform-dependant lines had to be **removed** from `/etc/X11/xorg.conf` :

```
Section "Device"
     Identifier "default"
     Driver "ati"
     ChipID 0x515a
EndSection
```

The CCMS configuration can be found on the CD-ROM: `/impl/ccms/` .

AIDA-NG

The version R09.2 V2.77.0002 of AIDA-NG was installed. Its configuration can be found on the CD-ROM:

- configuration of the solution: `/impl/aida/aida.*` ;
- configuration of the communication partner: `/impl/aida/partner.*` .

CADAS-ATS

The version V1.5.00.0014 of CADAS-ATS was installed. Its configuration can be found on the CD-ROM: `/impl/cadas/cadas.*` .

CADAS-IMS

The installed CADAS-IMS is based on the following components:

- main modules: V2.1.09.0006 (Fiji Project);
- map server: V2.1.10.0005 (Zimbabwe Project);
- CCMS High Availability: V1.3.78.

A basic configuration was set up.

4.3 Compliance with the requirements

With this implementation, following requirements are fulfilled:

Requirement	Summary	Justification of the compliance
2.3.1.1.1.(2)	Synchronisation with NTP servers.	- Each virtual machine can use NTP independently from the others. [VV10]

2.3.1.1.2.(4)
2.3.1.1.2.(6)
2.3.1.1.2.(8)
	Automatic switchover upon detection of a fault.	- If one (or several) subsystem or virtual machine fails, the system detects it and reacts as if there were no virtualization.
		- Possibilities and configurability of the system are unchanged.

2.3.1.1.7.(1)	2 AFTN/AMHS servers in main/hot standby configuration. RAID 1 or 5.	- RAID 5 on both hardware platforms.
2.3.1.1.7.(2)	UPS with a capacity of one hour.	- The power consumption is lower than in a traditional solution (see G Savings due to the project).
2.3.1.2.(1)	Use of HP ProLiant DL380 G6.	- More RAM, 2 processors and more NICs.

2.3.1.2.(3)
2.3.1.2.(4)
	Usual guest operating system without recompilation of the kernel.	- Fedora Core 11 32-bits usually delivered by Comsoft.

2.3.1.2.(5)	No software modification except on configuration components.	- The CCMS and CNMS may need to be extended.
2.3.1.2.(6)	Extensive support for new components.	- VMware ESXi benefits from a large support (see chapter 3.4.1 VMware Infrastructure).

2.3.2.(1)
2.3.2.(2)
	Savings on purchase, energy consumption, assistance and maintenance.	- See appendix G Savings due to the project.

2.3.3.(2)	Version of each software element mentioned.	- See previous parts of chapter 4 Implementation.

Table 4-1: Fulfilled requirements

Note: The compliance of this implementation with some requirements is based on the fact that the current Comsoft solutions also fulfil these requirements.

Some modifications would be needed to completely fulfil the following requirements:

Requirement	Summary	Lack

2.3.1.1.2.(1)
2.3.1.1.2.(2)
2.3.1.1.2.(3)
2.3.1.1.2.(10)
	Monitor and log the health state of all system components.	- The CNMS should be extended to be able to monitor and log the health of VMware ESXi.

2.3.1.1.3.(3)	Values in the offer based on existing operational systems.	- No operational system using virtualization exists yet.
2.3.1.1.3.(6)	Detail the procedure for rapid system re-installation.	- At least the CCMS Administrator's Guide [A14] shall be updated.

Table 4-2: Requirements not fully fulfilled yet

5 Verification and Validation

5.1 Hard disk latency

This test aims to easily **verify** that the problem exposed in chapter 2.1.1 Comsoft products, which is the **reason of this study**, has been **solved**. The RSS shouldn't be disturbed by an application running in another virtual machine, even if this application uses the hard disk intensively.

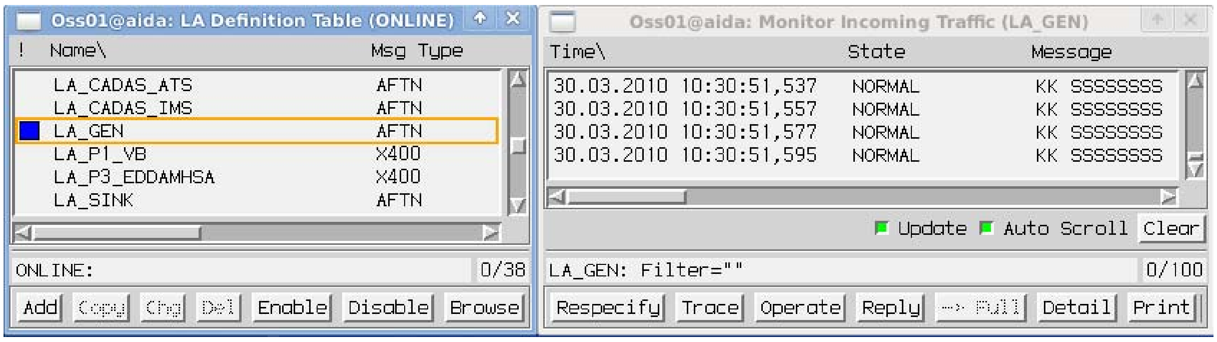
Part 1

Appendix H Health of an AIDA-NG Recording Sub-System shows the different ways of detecting when an RSS suffers from hard disk latencies.

Several instances of the test program presented in chapter 3.5.1.3 Hard Disk Drive have been run in the same virtual machine as RSS 1 (AIDA-NG 1) with the following command:

```
./loop.sh -f /tmp/test<N>
```

At the same time, a *message generator* generating 50 AFTN messages per second and a *traffic monitor* have been started:



Oss01@aida: LA Definition Table (ONLINE)

!	Name\	Msg Type
	LA_CADAS_ATS	AFTN
	LA_CADAS_IMS	AFTN
🟦	LA_GEN	AFTN
	LA_P1_VB	X400
	LA_P3_EDDAMHSA	X400
	LA_SINK	AFTN

ONLINE: 0/38

Oss01@aida: Monitor Incoming Traffic (LA_GEN)

Time\	State	Message
30.03.2010 10:30:51,537	NORMAL	KK SSSSSSSS
30.03.2010 10:30:51,557	NORMAL	KK SSSSSSSS
30.03.2010 10:30:51,577	NORMAL	KK SSSSSSSS
30.03.2010 10:30:51,595	NORMAL	KK SSSSSSSS

LA_GEN: Filter="" 0/100

Figure 5-1: Message generator and traffic monitor

The consequences were:

- With one instance, the RSS displayed some difficulties in `rsdb1times.txt`;
- With five instances, some manager warnings were generated and the traffic monitor froze;
- With 20 instances, the RSS went to maintenance.

Note: 20 instances not only make the hard disk busy but also use 100 % of the guest CPU.

We now have found a set of applications that makes the RSS crash. The goal of the next part is to check that if this set of applications is run in another virtual machine, the RSS doesn't crash.

Part 2

We ran at the same time:

- 30 instances in the virtual machine CADAS-ATS 1;
- 30 instances in the virtual machine CADAS-IMS 1.

RSS 1 didn't show any problem and the traffic monitor stayed fluid, which means that even the graphical interface had no problem: **two virtual machines were overloaded but the server wasn't.**

Comsoft's problem is solved.

5.2 Test scripts

In this chapter tests are run in order to prove that the remaining requirements are fulfilled. These test scripts are adapted from the official Comsoft test scripts. [A20] [A21]

Requirement	2.3.1.1.2.(5): (CSS-)Switchover in less than ten seconds.	
Test description	The duration of a switchover depends on the number of external lines (SNL boards). This test compares the duration of a switchover with no external line on our solution with the duration of a switchover on a classical solution (without virtualization) with no external line.	
Prerequisites	None.	
Procedure		**Expected Results/Observations**
1)	Make sure than both AIDA-NG machines have a satisfying time synchronisation with NTP: open a diagnosis dialog in an OSS and go to `/System/Supervision/Ntpd/`.	The values in the column "Deviation (ms)" are lower than 10.
2)	Perform five switchovers.	
3)	Open an event retrieval dialog and look for the events "CORE: OPERATIONAL-" and "OPERATOR: INITIATED CORE SWITCHOVER".	The time difference between two events gives the duration of a switchover.
4)	Perform steps 1 to 3 on a classical solution.	
Results	On both solutions, the duration of a switchover is 1.2 seconds (varying between 1.1 and 1.3 seconds).	
Comments	No difference between our solution and a classical one. Since a classical solution fulfils this requirement, **our solution fulfils this requirement**.	

Requirement	2.3.1.1.2.(7): Pending messages restored in less than five minutes.	
Test description	Pending message lists (PML) are filled and a switchover is performed.	
Prerequisites	- An AFTN message generator exists: its messages are routed to the AFTN mailbox. - An AMHS message generator exists: its messages are routed to a disconnected P3 LA (communication partner). - The legacy PML maximum size is 20 MB. - The X.400 PML maximum size is 20 MB.	
Procedure		**Expected Results/Observations**
1)	Launch the AFTN message generator until the legacy PML length is excessive: open an event monitor.	An event "ROUTING: PML LENGTH EXCESSIVE" occurs.

2) Launch the AMHS message generator until the X.400 PML length is excessive: open an event monitor.	An event "X.400: PML LENGTH EXCESSIVE" occurs.
3) Perform five switchovers.	
4) Open an event retrieval dialog and look for the events "CORE: OPERATIONAL+" and "OPERATOR: INITIATED CORE SWITCHOVER".	The time difference between two events gives the duration of the restoration.
Results	The duration of the restoration is 4.4 seconds (varying between 3.8 and 5.0 seconds).
Comments	**Our solution fulfils this requirement**.

Requirement	2.3.1.1.2.(9): Re-initialisation in less than ten minutes.	
Test description	Time needed to completely start a server and its software components.	
Prerequisites	- One of the servers is powered off. - The virtual machines of the other server are used to monitor the status of AIDA-NG (by means of an OSS), CADAS-ATS (by means of an Administration Terminal) and CADAS-IMS (by means of the Pacemaker Management GUI).	
Procedure		**Expected Results/Observations**
1) Start a stopwatch and power on the server.		
2) When the status of AIDA-NG, CADAS-ATS and CADAS-IMS are all green, stop the stopwatch.		The elapsed time gives the duration of the re-initialisation.
3) Perform steps 1 and 2 five times.		
Results	The duration of the re-initialisation is 5 min 55 s (+/- 10 s). AIDA-NG is the system which needs most time to re-initialise.	
Comments	**Our solution fulfils this requirement**.	

Requirement	2.3.1.1.4.(1), 2.3.1.1.4.(2), 2.3.1.1.4.(3), 2.3.1.1.4.(4), 2.3.1.1.4.(8), 2.3.1.1.4.(9): - Average message input rate of 20 messages per second. - Either 20 AFTN messages or 20 AMHS messages or a mixed message input. - Any combination of addresses. - Average message text size of 1 500 bytes, minimum message text size of 100 bytes, maximum message text size of 15 000 bytes. - Input-output ratio of 1:2. - No accumulation within the system. - These requirements concern only AIDA-NG. CADAS-IMS is not necessarily designed to handle this throughput.

Test description	- A communication partner sends 20 AFTN messages and 20 AMHS messages per second to AIDA-NG over TCP. (4 different destination addresses for AFTN and for AMHS) - The message sizes are: 25 % x 100 bytes; 5 % x 15 000 bytes; 20 % x 1 000 bytes; 50 % x 1050 bytes. - AIDA-NG sends two copies of each message back to the communication partner over TCP. - A copy of all messages is also sent to CADAS-ATS (even if this isn't part of the requirement). - A copy of the AFTN messages which have a length of 1 000 bytes is also sent to CADAS-IMS (even if this isn't part of the requirement). CADAS-IMS then receives 4 messages per second. CADAS-IMS doesn't support AMHS.	
Prerequisites	- Appropriate message generators on the communication partner. - Appropriate routing entries are configured on both sides. - Appropriate message sinks are configured on both sides.	
Procedure		**Expected Results/Observations**
1) Start all message generators on the communication partner. After one full minute, stop them.		
2) Check via statistics (`/Routing/LA/`) on the communication partner that the specified load has been sent (1 200 AFTN messages and 1 200 AMHS messages) and received (2 400 AFTN messages and 2 400 AMHS messages).		
3) Run the load again during at least one hour and check via statistics (`/Routing/LA/`) on the tested AIDA-NG that the system is not building queues.		The column "pending messages" doesn't grow.
Results	Passed.	
Comments	**Our solution fulfils these requirements**. Note that the maximum incoming throughput of our CADAS-IMS is 8 messages per second. It isn't possible to cover all possibilities (all combinations of addresses, etc.) but this test was more severe than the requirements (total of 40 messages per second instead of 20).	

Requirement	2.3.1.1.4.(6): Message transfer time within the system shorter than one second.	
Test description	Same as the previous test.	
Prerequisites	Same as the previous test.	
Procedure		**Expected Results/Observations**
1) Run the load defined in the previous test.		
2) Check via a *Traffic Transit Time Calculation* dialog on the tested system that the transmission time for the AFTN messages received from the communication partner and sent back to it is lower than one second.		The values "Transm. Unqueued / Maximum (ms)" and "Transm. Queued / Maximum (ms)" are lower than 1000.
3) Same verification for AMHS messages.		The value "Transm. Unqueued / Maximum (ms)" is lower than 1000.

Results	The highest time is 459 ms. The average is lower than 50 ms.
Comments	**Our solution fulfils this requirement**.

Requirement	2.3.1.1.4.(10).Part 1: AIDA-NG's graphical interface response time under average load lower than 500 ms.	
Test description	Same as the previous test.	
Prerequisites	Same as the previous test.	
Procedure		**Expected Results/Observations**
1) Run the load defined in the previous test.		
2) Browse in the graphical interface.		No noticeable response time.
Results	Passed.	
Comments	**Our solution fulfils this requirement**.	

Requirement	2.3.1.1.4.(7), 2.3.1.1.4.(10).Part 2: Process a peak load which is three times the average load specified for previous tests. AIDA-NG's graphical interface response time lower than one second.	
Test description	Same as the previous test, but: - All rates are multiplied by 2 (previous rates were already twice higher than the requirement). - No message sent to CADAS-IMS.	
Prerequisites	Same as the previous test.	
Procedure		**Expected Results/Observations**
1) Start all message generators on the communication partner. After one full minute, stop them.		
2) Check via statistics (`/Routing/LA/`) on the communication partner that the specified load has been sent (2 400 AFTN messages and 2 400 AMHS messages) and received (4 800 AFTN messages and 4 800 AMHS messages).		
3) Run the load again during at least one hour and check via statistics (`/Routing/LA/`) on the tested AIDA-NG that the system is not building queues.		The column "pending messages" doesn't grow.
4) Browse in the graphical interface during the test.		No noticeable response time.
Results	Passed.	
Comments	**Our solution fulfils these requirements**.	

Requirement	2.3.1.1.4.(5), 2.3.1.1.4.(6): Simultaneous transformation of 20 AFTN to AMHS messages per second and 20 AMHS to AFTN messages per second. Transfer time within the system shorter than one second.
Test description	- A communication partner sends 20 AFTN messages and 20 AMHS messages per second to AIDA-NG over TCP. Each message is 1 500 bytes long. - AIDA-NG converts each AFTN message into an AMHS message and sends it back to the communication partner. - AIDA-NG converts each AMHS message into an AFTN message and sends it back to the communication partner.
Prerequisites	Same as the previous test.
Procedure	**Expected Results/Observations**
1) Start all message generators on the communication partner. After one full minute, stop them.	
2) Check via statistics (`/Routing/LA/`) on the communication partner that the specified load has been sent (2 400 AFTN messages and 2 400 AMHS messages) and received (4 800 AFTN messages and 4 800 AMHS messages).	
3) Run the load again during at least one hour and check via statistics (`/Routing/LA/`) on the tested AIDA-NG that the system is not building queues.	The column "pending messages" doesn't grow.
4) Check via a *Traffic Transit Time Calculation* dialog on the tested system that the transmission time for the AMHS messages received from the communication partner and sent as AFTN messages back to it is lower than one second.	The values "Transm. Unqueued / Maximum (ms)" and "Transm. Queued / Maximum (ms)" are lower than 1000.
5) Same verification for the other direction.	The value "Transm. Unqueued / Maximum (ms)" is lower than 1000.
Results	The highest time is 229 ms. The average is lower than 60 ms.
Comments	**Our solution fulfils these requirements**.

Requirement	2.3.1.1.5.(3): Automatic switchover upon detection of a fault. Pending messages are not lost. **Note:** the duration of a manual switchover when pending message lists are full has already been measured in a previous test for the requirement 2.3.1.1.2.(7) and is lower than 5 seconds.
Test description	Pending message lists (PML) are filled and the physical hardware platform where the operational CSS runs is powered off.
Prerequisites	- An AFTN message generator exists: its messages are routed to the AFTN mailbox. - An AMHS message generator exists: its messages are routed to a disconnected P3 LA (communication partner). - The legacy PML maximum size is 20 MB. - The X.400 PML maximum size is 20 MB.

Procedure	**Expected Results/Observations**
1) Launch the AFTN message generator until the legacy PML length is excessive: open an event monitor. Note the number of pending messages.	An event "ROUTING: PML LENGTH EXCESSIVE" occurs.
2) Launch the AMHS message generator until the X.400 PML length is excessive: open an event monitor. Note the number of pending messages.	An event "X.400: PML LENGTH EXCESSIVE" occurs.
3) Open an OSS on the platform where the standby CSS is running.	
4) Remove the power cables of the hardware platform running the operational CSS.	The standby CSS becomes operational after a few seconds and the number of pending messages is still the same.
Results	The opened OSS detected that the operational CSS was gone after 10 seconds (same as for a classical solution). No message has been lost.
Comments	**Our solution fulfils this requirement**.

Requirement	2.3.1.1.6.(1): Capacity of CADAS-ATS servers to handle 50 terminals at the same time generating a total of 20 messages per second.
Test description	Starting 50 terminals to generate traffic would require a great number of workstations and technical experts. We used instead a configurable simulator ("Test Client") already developed by Comsoft for this purpose.
Prerequisites	- 50 different users exist. - The test client is configured to make each user generate messages with a period of 2.5 seconds (stricter than the requirement).
Procedure	**Expected Results/Observations**
1) Install the test client on one of the CADAS-ATS virtual machines.	
2) Start it and monitor the destination mailbox: `$ java -jar testclientall.jar test.xml`	The number of pending messages is incremented by 1 200 every minute, as soon as all users are logged in (see the output of the test client).
Results	Passed.
Comments	**Our solution fulfils this requirement**. The file `test.xml` is available on the CD-ROM: `/impl/cadas/test_cl/`.

5.3 Untested compliance

The compliance with the following requirements couldn't be tested within the given time:

Requirement	Summary	Comments
2.3.1.1.3.(1)	Availability 24 hours per day and 7 days per week.	- No component of the solution needs to be regularly restarted. - However the solution should be tested during one month under real load.
2.3.1.1.3.(2)	Availability greater than 99.999 %.	- The solution should be tested during one month under real load.
2.3.1.1.3.(7)	Detail the MTBF and the MTTR.	- No unexpected failure has been observed, so it was impossible to determine the MTBF. - The MTTR requires the determination of all types of failures to be repaired. - Those values could have been calculated if VMware published its own values concerning the availability of ESXi.

Table 5-1: Unproven compliance

The solution has been **used and tested during one month** (however not always under real load) and showed a **high stability** comparable to a classical solution (without virtualization). There is no reason to think that those requirements may not be fulfilled.

5.4 Other requirements

Following requirements are finally fulfilled:

Requirement	Summary	Justification of the compliance

2.3.1.1.8.(1)
2.3.1.1.8.(2)
2.3.3.(1)
	Demonstration of the compliance to the requirements and demonstration of the capabilities by a practical presentation including all major blocks of the solution. Traceability between each requirement and each justification.	- All requirements are exposed in chapter 2.3 Requirements.
		- The compliance to each requirement has been shown in chapters 3.3 Compliance with the requirements, 4.3 Compliance with the requirements and 5 Verification and Validation.
		- The requirements which are not fully fulfilled yet are listed in chapters 4.3 Compliance with the requirements and 5 Verification and Validation.
		- The proof of the compliance is based on the complete evaluation of VMware ESXi in chapter 3.5.1 VMware ESXi.
		- The main blocks of the solution are presented in chapters 3.2 New design and 4.1 Architecture.
		- A real solution has been built and tested.

Table 5-2: Fulfilled requirements

6 Summary and Outlook

Comsoft needed a solution for server consolidation with the possibility to distribute equally resources among its soft real-time applications and the guarantee that the time needed for an application to get a resource is limited. They ordered a study based on server virtualization.

We solved their problem by designing and implementing a solution based on VMware ESXi. This solution provides a satisfying isolation between applications, high predictability and limited latencies, with low performance degradation. The validity of the solution has not only been tested but also proved by means of precise and strict statistic indicators. Moreover this study is solidly founded on a significant amount of external documents and sources. All aspects of this study are precisely documented and can be reproduced.

Apart from its contribution to Comsoft, this study brings also a scientific contribution. It defines a series of indicators and tests which can be used to precisely evaluate predictability, performance and isolation of virtualization products in a soft real-time context. More generally this study can be used to evaluate any level of a computer system (e.g. an operating system or an application framework). We encourage anyone who wants to extend and reuse this methodology.

The implemented solution fulfils almost all requirements (*availability*, *mean time between failures* and *mean time to repair* still have to be determined) and could be sold. The next steps for Comsoft are as follows:

- The application monitoring the health of the systems (CNMS) should be extended to monitor the health of the virtualization layer.
- Installation procedures have to be defined and the corresponding documentation has to be written. Maintenance procedures have to be updated.
- A configuration tool for ESXi could be developed in order to replace vSphere Client, so that no Windows workstation has to be delivered.

The current trend in virtualization is to improve mean performance by implementing more and more optimizations, thus reducing the predictability of the hypervisors and their schedulers. This may represent a danger for Comsoft since VMware ESXi may become less predictable in the future. The tests defined in this study should be rerun after each major change in ESXi.

At the same time, Comsoft may re-evaluate other virtualization products: they may also become valid in the future.

Finally, care has to be taken not to consider VMware ESXi as the best virtualization solution on the market. Several other products have been excluded at the beginning of this study due to requirements specific to Comsoft. Their predictability or real-time properties may be actually better than ESXi.

Appendix A Implementing full virtualization

This appendix gives an overview of the different techniques used to implement a full virtualization product and their respective complexities.

Problem

Nowadays, every architecture (except some embedded systems' architectures) generally offers at least two execution modes: *kernel mode* and *user mode*. Let's considerer first computers with only one execution unit (see chapter 3.5 Evaluation). On such a computer, only one execution path or code sequence can be run at a time. That means that if an application is running, the operating system is not: contrary to what one might think, **a running application is not monitored by the operating system**. But an application runs in *user mode*. In this execution mode, some processor instructions are forbidden (a *protection fault* exception will be thrown if an application tries to execute such an instruction). Those *protected instructions* can only be run in *kernel mode*. The operating system will get control of the computer back on following events:

- *interrupt*: event triggered by a component[1] (e.g. the keyboard indicates that a key was pressed) independently from the instruction being executed at that moment (the interrupt's cause is not the executed code). Moreover one timer can trigger periodically an interrupt to guaranty that the operating system is regularly in control (which means the operating system gets the execution unit).
- *exception*: event that is the result of the (usually abnormal) execution of one instruction (e.g. division by zero or **protection fault**).
- *system call* (a.k.a. *syscall*): special instruction used to jump to a service provided by the operating system. For instance if an application wants to write a character on a device, it cannot do it directly (it would throw a protection fault exception[2]) but has to perform a system call to let the operating system's kernel (running in *kernel mode*) do it. A system call could be considered as an instruction always throwing an exception, but since an exception is usually unwanted, we usually distinguish system calls from exceptions.

When one of those three events happens, the processor performs following operations[3]:

- the context of the running application (registers' content, stack, …) is saved;
- the execution mode is set to *kernel mode*;
- the processor jumps to the handler corresponding to this event (interrupt, exception or system call);
- the handler (and thus the operating system) is run;
- the processor jumps back to the application (or to another one, depending on the scheduling policy): the context of the application is restored and the execution mode is set to *user mode*.

As you can see, the operating system's kernel runs in kernel mode and has full access to hardware. This is usually wanted: the role of the operating system is to decide which application receives which hardware resource and the operating system can assume it is the only decision-maker on board concerning the entire hardware.

[1] Internal or external to the processor.
[2] The standard reaction of an operating system to such exception is the destruction of the application.
[3] You should be aware that this is a simplification to understand the principles. You can find more details on this topic in *Operating Systems* [C7] and *Understanding the Linux Kernel* [C8].

However this behaviour is not acceptable if several (guest) operating systems are running. In this configuration, the decision-maker is the *hypervisor*, which should be called each time a decision concerning hardware allocation to virtual machines has to be made.

But how can this be achieved without modifying the guest operating system?

Solution 1: Hardware Support

We need a way to:

- detect when a guest tries to access hardware while executing an instruction in kernel mode;
- suspend the execution of this instruction;
- put the hypervisor in control, so that it can perform some operations, decide whether the guest is allowed to perform this access or not, and resume the guest's execution or not, according to the hypervisor's scheduling policy.

In other words, instructions which usually don't throw any exception if executed in kernel mode shall now throw an exception if executed by a guest, in order to make the processor jump to an exception handler set up by the hypervisor. Therefore two additional modes are needed:

- *guest mode*: the code of a guest (virtual machine) is run in this mode;
- *hypervisor mode*: the code of the hypervisor is run in this mode.

This behaviour is also known as *trap-and-emulate*. [V6]

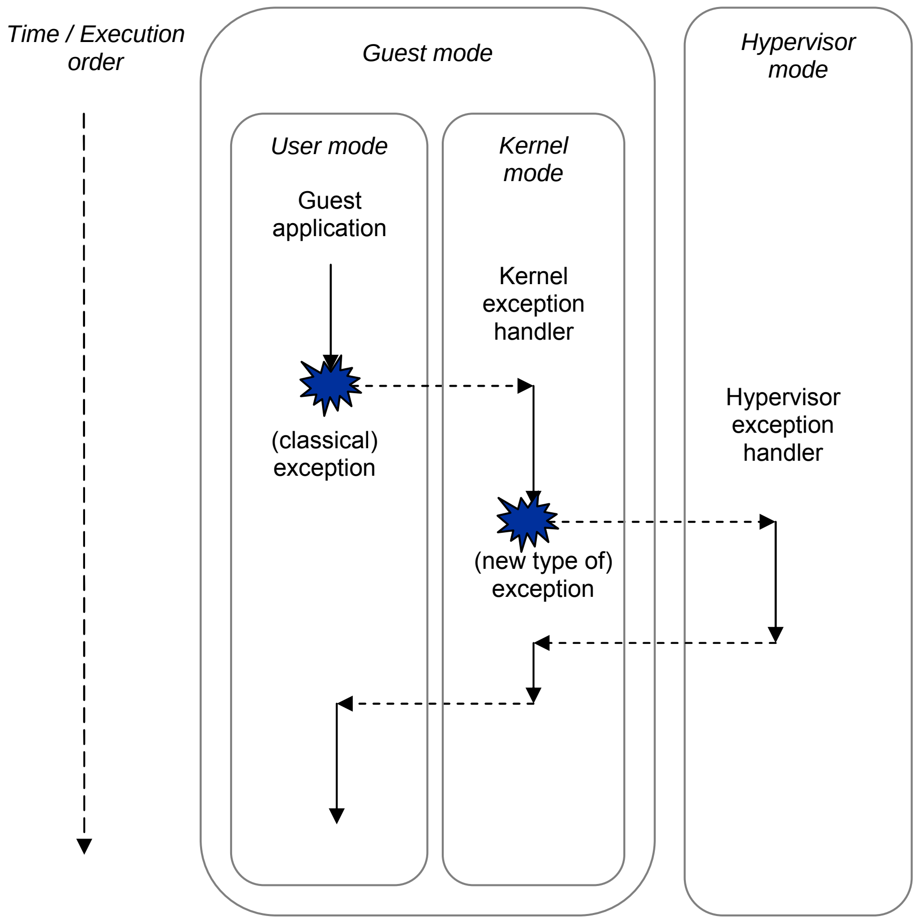


Figure A-1: Relations between the newly defined modes and exceptions

Solutions based on this model have been developed:

- Intel developed Intel VT-x, the virtualization technology on the x86 platform. [V9]
- AMD developed AMD-V (AMD Virtualization).

In the Intel solution, the processor support for virtualization is called VMX (Virtual Machine eXtension). The hypervisor is called *Virtual Machine Monitor*. The hypervisor mode is called *VMX root operation* and the guest mode is called *VMX non-root operation*. Transitions between those modes are called *VMX transitions*:

- a VMX transition from the VMX non-root operation to the VMX root operation is a *VM exit*;
- a VMX transition from the VMX non-root operation to the VMX root operation is a *VM entry*.

Some instructions, if executed in VMX non-root operation (i.e. executed by a guest) will trigger a VM exit, allowing the Virtual Machine Monitor to control hardware resources.

There is no way for software (e.g. no special instruction or register) to know if it is being executed in VMX non-root operation or in VMX root operation. Thus an operating system can't know if it is being executed on a virtual machine. You will see in part Solution 2: Protection Ring 1 that this is an important point.

More details can be found in the Intel documentation [C2] or in *Intel® Virtualization Technology: Hardware support for efficient processor virtualization* [V5].

Solution 2: Protection Ring 1

Hardware support for virtualization doesn't exist on every computer and is actually quite new on the x86 architecture (Intel-VT is available since 2005 [V9]).

In this architecture, kernel mode and user mode are implemented by means of four *privilege levels* (a.k.a. *protection rings*). An operating system should run in the highest privilege level (ring 0, representing the kernel mode) whereas code not to be trusted (e.g. applications) should run in the lowest privilege level (ring 3, representing the user mode). Privilege levels are not only assigned to code segments, but also to data segments and devices. Following rules apply:

- A program (i.e. a peace of code) can only access data and devices from its own ring or from a ring with lower privilege. Thus an application cannot access the operating system's data objects.
- A program can only call another program from its own ring or from a ring with more privilege. Thus a program can only call one that is more trusted. That is why an application can ask for a service from the operating system's kernel by means of a system call.
- Anything that goes against those rules triggers a *General Protection Fault* exception.

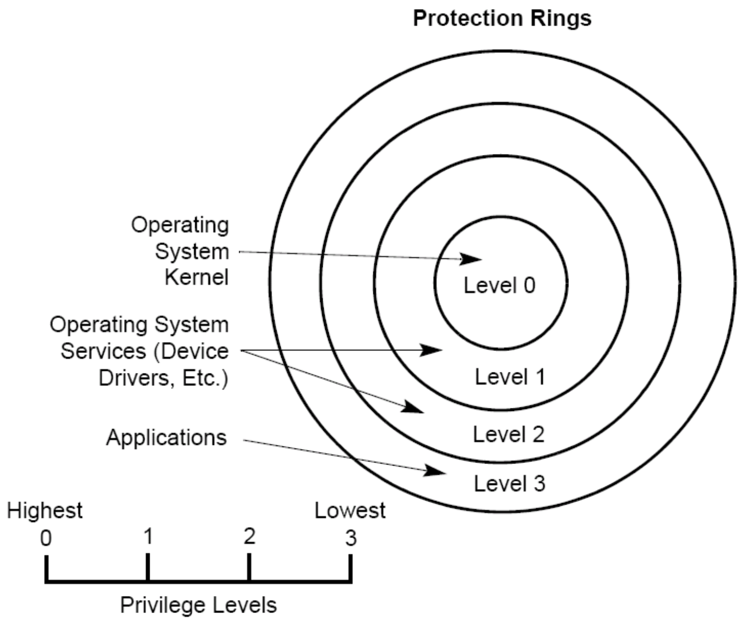


Figure A-2: Intel's Protection Rings (from the Intel Documentation [C1])

More details on the privilege levels on the x86 architecture can be found in the Intel Documentation [C1].

It follows from the previous rules that the operating system's kernel has access to the entire hardware. Running applications in the protection ring 3 (user mode) and assigning protection rings 0, 1 or 2 to devices ensure that no application will be able to directly access the hardware.

Whereas protection rings 1 and 2 should be used for device drivers, most operating systems only use rings 0 and 3 to implement kernel and user modes. One idea to achieve virtualization was to move the guest kernel's code into protection ring 1 (which is not used) and to put the hypervisor's code and devices which have to be shared among virtual machines in protection ring 0. Thus each time a guest's kernel tries to access a device under control of the hypervisor, a General Protection Fault exception is thrown and the processor jumps to the corresponding exception's handler of the hypervisor.

Although this solution seems perfect and we think we might have solved the problem exposed in part Problem, it is not. It is possible for *any* application to know in which protection level it is being executed: the instruction or register indicating the *current protection level* is not protected.[1] A guest kernel running the following code (C++) in a virtual machine will crash:

```
// Crashes if the condition is false:
assert(myRing == 0);
```

The main difference between this solution and the previous one (hardware support) is that in this solution, the guest can know if it is running in a virtual machine or not. This contradicts one of the formal requirements for virtualizable architectures from Gerald J. Popek and Robert P. Goldberg:

> "Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies." [V2]

And this problem is real since common operating systems (e.g. Linux and Windows) actually make use of such problematic instructions. Thus the solution presented in this chapter ("protection ring 1") is **not a valid solution**.

Interesting information on this topic can be found in:

- *Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor* [V3];
- *Running multiple operating systems concurrently on an IA32 PC using virtualization techniques* [V4];
- *Intel® Virtualization Technology: Hardware support for efficient processor virtualization* [V5];
- *A Hardware Architecture for Implementing Protection Rings* [C10];
- *Das Virtualisierungs-Buch* [V1].

Note: The *hypervisor mode* created in the *hardware support* solution is often referred as "ring -1", in comparison to the four existing protection rings presented in this chapter.

Solution 3: Dynamic Recompilation

Dynamic recompilation (or *dynamic binary translation*) is an optimization of *emulation*. The basic structure of an emulator is, as already mentioned in chapter 2.1.2.2 Virtualization Techniques, the same as an interpreter's. If you ignore interrupt handling, the code should look like this (C++):

```
// also known as Program Counter:
unsigned long instructionPointer = FIRST_INSTR_OF_THE_GUEST;

for (;;)
{
    // fetch the next instruction:
    instruction i = fetch(instructionPointer);
```

[1] More details in the Intel Documentation [C1] [C2].

```
    // execute it:
    switch (i.opCode())
    {
        case OP_CODE_1: // example: ADD
            // implementation (updates instructionPointer)
            break;

        case OP_CODE_2:
            // implementation (updates instructionPointer)
            break;

        // ...

        case OP_CODE_N:
            // implementation (updates instructionPointer)
            break;
    }
}
```

Since in our case, the architecture of the virtual machine is the same as the host's, we can execute each instruction directly on the host processor (there is no need to re-write an implementation for each instruction). Moreover, in this case we are developing a hypervisor which handles **several** guests. Thus the hypervisor should decide periodically which guest should be run, according to its scheduling policy:

```
// also known as Program Counter:
unsigned long instructionPointer[NB_GUESTS];

initialize(instructionPointer);

int guestToRunNext = 0;

for (;;)
{
    // fetch the next instruction:
    instruction i = fetch(instructionPointer[guestToRunNext]);

    // execute i directly on the processor
    // (updates instructionPointer[guestToRunNext])

    // decide which guest will run next:
    guestToRunNext = scheduler();
}
```

But as we said, some instructions are not safe and should be intercepted, to prevent the guest from directly accessing the hardware without the hypervisor's permission. We have to mix the two previous pieces of code:

```
// also known as Program Counter:
unsigned long instructionPointer[NB_GUESTS];

initialize(instructionPointer);

int guestToRunNext = 0;

for (;;)
{
    // fetch the next instruction:
    instruction i = fetch(instructionPointer[guestToRunNext]);

```

```
     // execute it:
     switch (i.opCode())
     {
          case OP_CODE_SAFE_1:
          case OP_CODE_SAFE_2:
          // ...
          case OP_CODE_SAFE_N:
               // execute i directly on the processor
               // (updates instructionPointer[guestToRunNext])
               break;


          case OP_CODE_UNSAFE_1:
               // implementation
               // (updates instructionPointer[guestToRunNext])
               break;

          case OP_CODE_UNSAFE_2:
               // implementation
               // (updates instructionPointer[guestToRunNext])
               break;

          // ...

          case OP_CODE_UNSAFE_N:
               // implementation
               // (updates instructionPointer[guestToRunNext])
               break;
     }

     // decide which guest will run next:
     guestToRunNext = scheduler();
}
```

This version of the emulator now works without any hardware support and is a little more powerful than a normal emulator, since we don't emulate *every* instruction, but only the ones which are *unsafe*. Dynamic recompilation is an evolution of this principle: instead of fetching and executing each guest instruction one by one, it fetches instructions block by block to improve speed. Then the block is parsed and each unsafe instruction is replaced by a jump to one of the hypervisor's handlers. After those replacements, the whole block is considered as safe and can be run directly by the processor:

```
// also known as Program Counter:
unsigned long instructionPointer[NB_GUESTS];
instructionBuffer buffer[NB_GUESTS];

initialize(instructionPointer);

int guestToRunNext = 0;

for (;;)
{
     // fetch the next instructions:
     fetch(instructionPointer[guestToRunNext], &buffer[NB_GUESTS],
           NB_INSTRUCTIONS_TO_BE_FETCHED);
```

```
    // parse and modify the buffer to make it safe:
    // (the implementation of makeSafe() should consist of a
    // "switch (opCode)" statement)
    makeSafe(&buffer[NB_GUESTS]);

    // execute it (updates instructionPointer[guestToRunNext])

    // decide which guest will run next:
    guestToRunNext = scheduler();
}
```

Of course the previous algorithm is a huge simplification of a real hypervisor based on dynamic recompilation. For instance, following aspects have been ignored:

- When letting a block run, the hypervisor actually loses control and the processor won't jump back to the hypervisor by itself after having executed the block. The `makeSafe()` function has to add this "jump back" instruction at the end of each block before letting it run.
- The hypervisor fetches several instructions (let's say 256) at a time (block by block), but we don't have the guarantee that those 256 instructions will be executed. The 10th instruction may be a *conditional jump* (`if` *`condition`* `then` *`jump`*) likely to jump to a place outside the block. First the `makeSafe()` function has to convert this instruction so that the control is not lost forever (if the processor jumps out of the block, the "jump back" instruction at the end of this block will never be executed and the hypervisor will never get control of the machine again). But then we face a performance problem: is a hypervisor parsing and converting 256 instructions in order to execute only ten of them efficient?

This last problem, encountered while implementing a *virtual processor* (i.e. the part of the hypervisor's code responsible for executing guest's instructions), is in fact already well known concerning the implementation of a *real microprocessor*. One set of solutions is named *Branch Prediction*. You can find a lot of explanations concerning this issue as well as all the issues encountered when developing a modern (real or virtual) microprocessor in *Computer Architecture* [C4] and *Mikrocontroller und Mikroprozessoren* [C5].

Once all those issues are solved by means of the same optimizations as those used in a real microprocessor or in a classical interpreter, a hypervisor using dynamic recompilation can be much faster than one fetching one instruction at a time.

More details on dynamic recompilation for virtualization can be found in:

- *Running multiple operating systems concurrently on an IA32 PC using virtualization techniques* [V4];
- *Introduction to Dynamic Recompilation* [V50];
- *An Emulator Writer's HOWTO for Static Binary Translation* [V51];
- *A Comparison of Software and Hardware Techniques for x86 Virtualization* [V6].

Comparison

Keith Adams and Ole Agesen (from *VMware*) wrote an interesting paper in 2006 [V6] dispelling the myth that a hardware-driven solution is necessarily faster than a software-driven one. At the time, *Intel-VT* and *AMD-V* solutions for hardware-supported virtualization were quite new (Intel-VT is available since 2005 [V9]) and they showed that:

- "Software and hardware VMMs both perform well on compute-bound workloads" [V6], which means native performances (without virtualization) are almost reached, since:
  - a software VMM (using *dynamic recompilation*) doesn't need to modify this type of instructions, which are *safe*;
  - those instructions don't trigger any *VM exit* with a hardware VMM (using *hardware support*).
- "For workloads that perform I/O, create processes, or switch contexts rapidly, software outperforms hardware" [V6], because:
  - "each guest page table write causes a VM exit" [V6] with a hardware VMM, which has a non-negligible cost;
  - *mode switches* are avoided with a software VMM by replacing *unsafe* instructions.
- "In two workloads rich in system calls, the hardware VMM prevails" [V6], because:
  - system calls can be executed natively with a hardware VMM;
  - they have to be replaced when using a software VMM.

This paper tends to show that at the time, a software VMM was lightly more efficient than a hardware one, though the authors hoped that the improvements announced by Intel and AMD may change this: AMD's "nested paging" and Intel's "EPT".

"In both schemes, the VMM maintains a hardware-walked 'nested page table' that translates guest physical addresses to host physical addresses. This mapping allows the hardware to dynamically handle guest MMU operations, eliminating the need for VMM interposition." [V6]

Hardware VMMs still have the advantage of being drastically simpler that software VMMs, thus reducing the number of potential bugs. Hybrid VMMs [V6] are VMMs trying to combine the advantages of both software and hardware VMMs. Paravirtualization (see chapter 2.1.2.2 Virtualization Techniques) may also provide a good optimization for hardware VMMs.

Appendix B CPU reservation on VMware ESXi

ESXi provides the possibility to define a *CPU reservation* for each virtual machine and for each pool (group of virtual machines). With this feature, you can assign for example 70 % of the CPU time to one virtual machine.

Again, this feature was tested on two different platforms:

- one with two Intel Xeon E5520 processors, with hyperthreading disabled;
- one with one Intel Xeon E5520 processor, with hyperthreading enabled.

Both platforms have 8 execution units, each unit having a frequency of 2 267 MHz. Thus ESXi considers that the total frequency is 8 x 2 267 = **18 136 MHz**.

To assign 70 % of the CPU time to one virtual machine, we have to define a *CPU reservation* of 70 % for this virtual machine. In ESXi however, CPU reservations are not defined with percentages but with MHz. So the reservation for this virtual machine will be 18 136 x 70 % = **12 696 MHz**.

Note: Of course, the sum of all reservations can't be greater than 18 136 MHz.

With one processor

There is obviously a **bug** in vSphere Client (the configuration tool) or ESXi for this hardware platform, since the **total** CPU reservation is limited to **7 137 MHz instead of 18 136 MHz**. Here are some different appearances of the bug:

- The maximum value of the CPU reservation for a virtual machine (vSphere Client / Settings / Resources / CPU / Reservation) is 7 137 MHz whereas the maximum value of the limitation (vSphere Client / Settings / Resources / CPU / Limit) is 18 136 MHz.
- The maximum value of the CPU reservation for a pool (vSphere Client / Settings / CPU Reservation) is 7 137 MHz. For the pools the maximum value of the limitation (vSphere Client / Settings / CPU Limit) is 7 137 MHz, which is not consistent with the previous point.
- In the tab "resource distribution" of the host, the CPU speed is 7 137 MHz.
- In "distribution of system resources" in the tab "configuration", you can configure the CPU reservation for ESXi. The default reservation is 227 MHz, but this value and the limitation can grow up to **9 066 MHz**. If you choose 9 066 MHz, you get an error dialog displaying "**vim.fault.InsufficientCpuResourcesFault**".

All those values are inconsistent. No document related to this problem has been found. We assigned 100 % of those 7 137 MHz to one virtual machine but we didn't observe any difference when running two virtual machines.

Note: the same problem was observed:

- with ESXi 4.0.0 Update 1, HP Customized, Build 208167 (special version for HP ProLiant)
- with ESX 4.0.0 Update 1 Build 208167 (last available version on 2010-02-04);
- with ESXi 3.5.0 Update 5 Build 207095 (last version of ESXi 3.5)
- after a firmware upgrade of the platform with HP Firmware Update CD 8.70 (last available version on 2010-02-04)

With two processors

With two processors without hyperthreading, the most disturbing part of the bug is gone:

- The maximum value of the CPU reservation for a virtual machine (vSphere Client / Settings / Resources / CPU / Reservation) is 16 201 MHz and the maximum value of the limitation (vSphere Client / Settings / Resources / CPU / Limit) is 18 128 MHz. (This small difference is probably due to some resources being kept free for ESXi.)
- For the pools the maximum value of the limitation (vSphere Client / Settings / CPU Limit) is 16 201 MHz too.
- In the tab "resource distribution" of the host, the CPU speed is 16 201 MHz.
- This time, the CPU reservation for ESXi can grow up to **18 133 MHz**. Again, if you choose this value, you get an error dialog displaying "**vim.fault.InsufficientCpuResourcesFault**".

We run the program described in the second part of chapter 3.5.1.1.1 Overhead on CPU-Bound applications on two virtual machines (A and B), with M = 10 000 and T = 16, and we assigned 100 % of the CPU to virtual machine A. The application in virtual machine B needed exactly twice as much time as the application in virtual machine A, which means that virtual machine A really had 100 % of the CPU at the beginning, although both applications actually "started" at the same time:

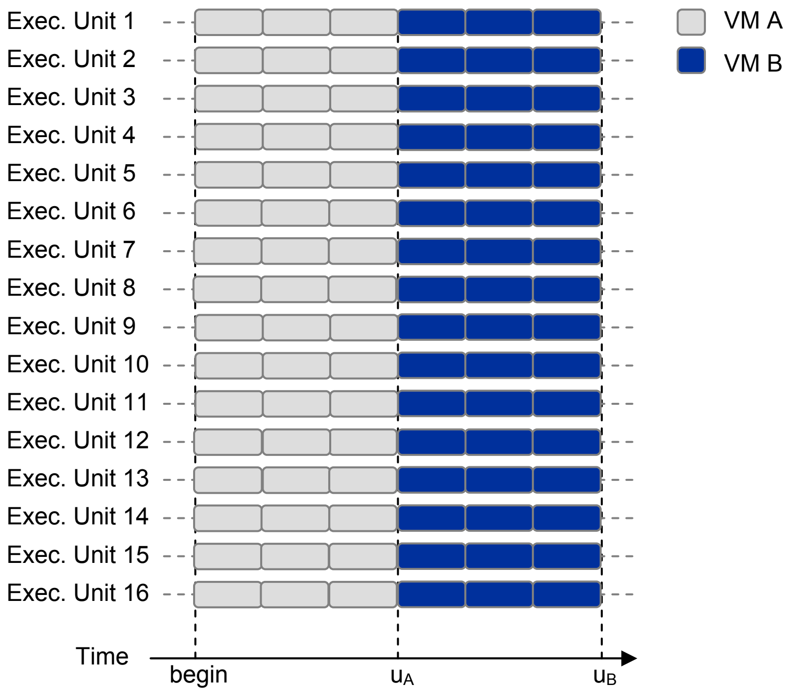


Figure B-1: CPU time distribution with a CPU reservation of 100 % for VM A.

We didn't test the predictability of this feature.

Appendix C Scheduling test on MS-DOS

The test exposed in chapter 3.5.1.1.5 Scheduling could have been developed on a *monotasking* system. Since this solution was explored and since it's hard to find documentation on this topic, this appendix exposes the different achieved steps for the interested reader.

We installed FreeDOS (free alternative to MS-DOS) [CR14] and DJGPP (version of GCC for MS-DOS) [CR15].

Test program

This example gets the current time from the CMOS RTC and the three PIT counters [VV10] and displays it. Details on how to program those PIT counters can be found in documents [C19] and [C20].

The assembly syntax used with GCC is called the AT&T syntax. The implementation of this example is available on the CD-ROM: `/eval/cpu/msd_test/` .

Conclusion

If you try to implement the test described in chapter 3.5.1.1.5 Scheduling, you will see that the frequency of PIT-0 is high enough, but its length (16 bits) is too short (it rolls over in a few milliseconds). So to calculate the elapsed time between two iterations of a loop, you need to compare values coming not only from PIT-0 but also from another timer with a lower frequency.

The CMOS RTC can't be used for this purpose because its frequency is too short: the time needed to make PIT-0 roll over is shorter than the CMOS RTC period. PIT-1 or PIT-2 could be used for this purpose, but are again too short. The only solution would be to use all of this at the same time:

- PIT-0 to determine the elapsed time,
- PIT-1 or PIT-2 to determine how many times PIT-0 rolled over, and
- CMOS RTC to determine how many times PIT-1 or PIT-2 rolled over.

It's possible but not convenient. This is why we chose the TSC timer to implement this test.

Concerning MS-DOS, our opinion is that it isn't as convenient and as easy to configure as Linux (in particular for networking and communication). This is why we chose a real-time operating system based on Linux to implement this test.

Appendix D Scheduling test on Fedora Core 11

Chapter 3.5.1.1.5 Scheduling presents a test to evaluate the scheduler of ESXi. As explained, it's important for this purpose to use a *hard real-time operating system*, because if the scheduler of the guest operating system isn't fully predictable, it isn't possible to evaluate the virtualization layer's scheduler.

This appendix shows the results of this test program run on a Fedora Core 11 without virtualization, to prove that this operating system was not adapted for this purpose.

When running the test 5 times, you can observe a significant difference between the best case and the worst case:

[0, 10[	[10, 100[	[100, 1K[	[1K, 10K[	[10K, 100K[	[100K, 1M[	[1M, 10M[	[10M, 100M[	[100M, 1G[	[1G, 10G[	[10G, 100G[	[100G, ∞[
0	999 330 307	637 911	30 727	808	247	0	0	0	0	0	0

Table D-1: Scheduling test on Fedora without virtualization, best case

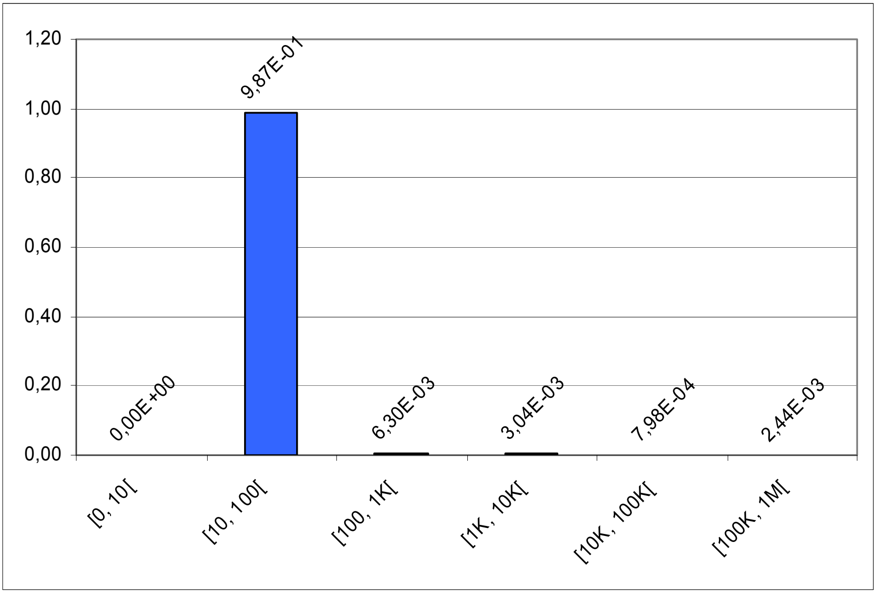


Figure D-1: Distribution of the influence on Fedora without virtualization, best case

[0, 10[	[10, 100[	[100, 1K[	[1K, 10K[	[10K, 100K[	[100K, 1M[	[1M, 10M[	[10M, 100M[	[100M, 1G[	[1G, 10G[	[10G, 100G[	[100G, ∞[
0	992 319 957	7 629 551	39 381	2 408	8 702	1	0	0	0	0	0

Table D-2: Scheduling test on Fedora without virtualization, worst case

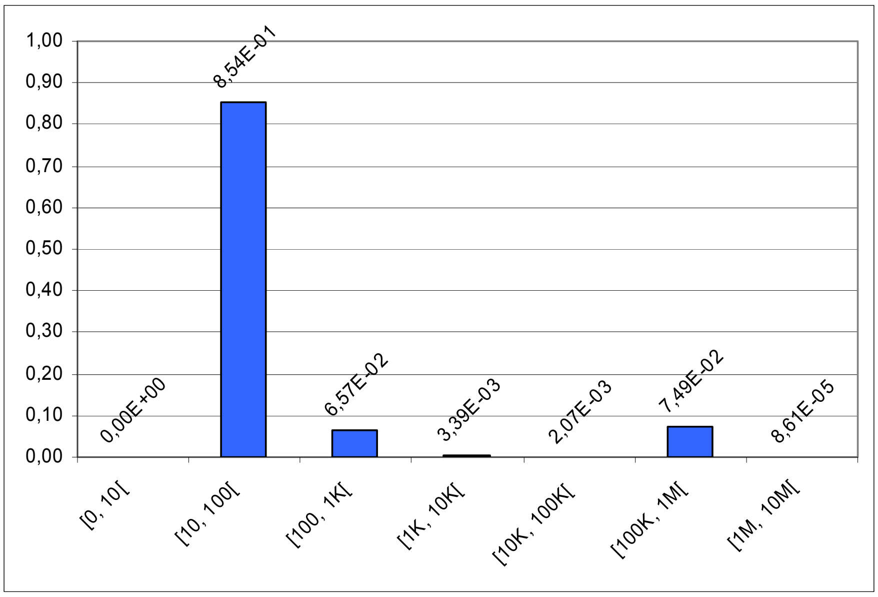


Figure D-2: Distribution of the influence on Fedora without virtualization, worst case

On the best case, $bound_{95\%}$ = 100 ticks = 44.1 ns, exactly the same as on Xenomai. In this case, the test process was obviously very rarely de-scheduled by the Linux scheduler.

On the worst case, $bound_{95\%}$ = 1 million ticks = 441 µs, which means that if a process wants to execute, it may have to wait for an execution up to 441 µs on this system. Of course this value is just an order of magnitude.

The variation between the best case and the worst case is really significant, whereas both experiments have been made in the same conditions. How can later variations induced by the virtualization level be detected if the reference test is so unstable?

Moving the mouse during the test for example can make the results even worse, which is not the case on Xenomai.

Appendix E Hard-disk latency reduction with ESXi

Other solutions than the one exposed in chapter 3.5.1.3 Hard Disk Drive have been explored in order to reduce the latency and isolate the different virtual machines. This appendix gives an overview of the results.

Distribution of the instances of the test program on different RAID channels

We tested the throughput and the latency when distributing the different instances of the test program over two RAID 5 groups, both connected to the same *RAID controller* but each of them connected to one *RAID channel*. The goal of this test is to determine if the use of two different RAID channels delivers guarantees.

Without virtualization, a high deviation of the writing throughput appeared when both channels were under high solicitation: we couldn't predict before launching the test program if the throughput would be 150 MB/s or 200 MB/s. However the observed latency stayed low: $\text{bound}_{95\%} \approx 250$ ms.

Over ESXi with two virtual machines, each of them using one different RAID group, the result was worse. When both channels were under high solicitation at the same time, one had a throughput of 142 MB/s and the other one only 20 MB/s, with a latency reaching 3.6 seconds. We couldn't predict before launching the test program which one of both virtual machines would suffer from a low throughput. Applying the throughput limitations defined in chapter 3.5.1.3.5 Improvement attempt solved the problem.

Therefore we prefer to consider that **using different RAID channels of the same RAID controller doesn't deliver any guarantee**.

Distribution of the instances of the test program on different RAID controllers

The same test has been run on different *RAID controllers*. This time we observed a perfect isolation and a low latency with and without virtualization.

Using different RAID controllers is a good way to guarantee a low latency, but it isn't adapted to this project since we only have two RAID controllers and we need to isolate at least three Comsoft products on the same hardware platform. Comsoft could buy additional RAID controllers (on PCI boards) but the layout of HP ProLiant DL380 G6 servers makes it impossible to insert additional hard disk drives. (The existing hard disk slots can't be used for this purpose: they are already connected to the provided RAID controllers.)

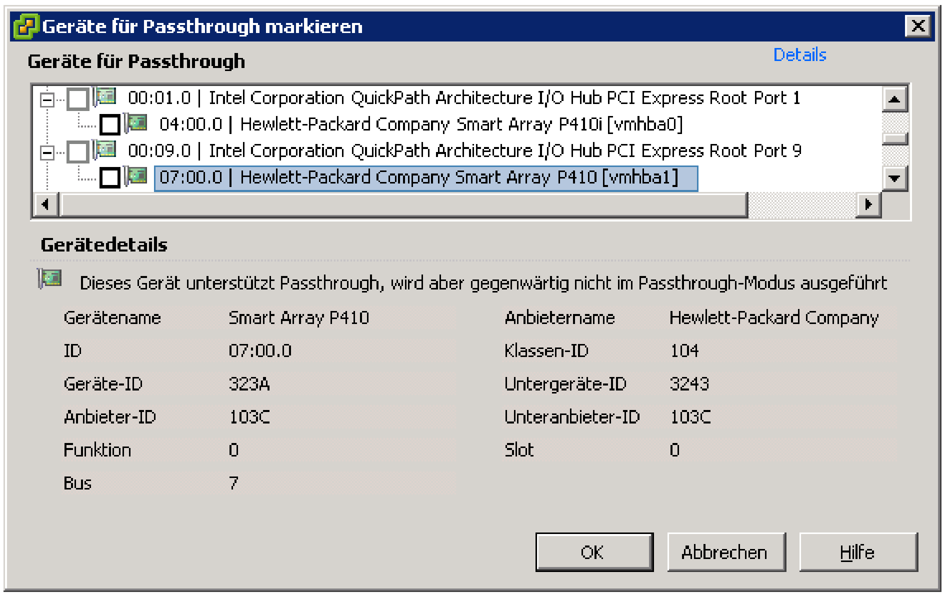
Based on the same principle, we could imagine using one distinct physical iSCSI device (Internet SCSI)[1] per virtual machine or a similar technique. VMware ESXi provides the possibility to create a virtual hard disk drive on an iSCSI device. [VV3] However this solution hasn't been explored. An external scheduler for storage virtualization delivering latency guarantees could then be used, as described in chapter 2.2 Related Work.

[1] Protocol for SCSI over TCP/IP.

VMDirectPath

VMDirectPath (or Passthrough) allows guest operating systems to directly access an I/O device, bypassing the virtualization layer. It can of course improve performance, but it prevents several virtual machines from sharing the same hardware device. [VV25]

To see the list of all devices supporting VMDirectPath, open vSphere Client / Configuration / Hardware / Advanced settings / Configure Passthrough :


Geräte für Passthrough markieren

Geräte für Passthrough

Details

00:01.0 | Intel Corporation QuickPath Architecture I/O Hub PCI Express Root Port 1
04:00.0 | Hewlett-Packard Company Smart Array P410i [vmhba0]
00:09.0 | Intel Corporation QuickPath Architecture I/O Hub PCI Express Root Port 9
07:00.0 | Hewlett-Packard Company Smart Array P410 [vmhba1]

Gerätedetails

Dieses Gerät unterstützt Passthrough, wird aber gegenwärtig nicht im Passthrough-Modus ausgeführt

Gerätename	Smart Array P410	Anbietername	Hewlett-Packard Company
ID	07:00.0	Klassen-ID	104
Geräte-ID	323A	Untergeräte-ID	3243
Anbieter-ID	103C	Unteranbieter-ID	103C
Funktion	0	Slot	0
Bus	7		

Figure E-1: VMDirectPath configuration dialog

Almost all PCI devices are supported for Passthrough. However figure E-1 shows that the different RAID channels can't be assigned, but only the RAID controllers, which would prevent us from creating more than two virtual machines. It's actually even worse: the first RAID controller can't be assigned to a virtual machine, since ESXi is installed on it.

More information about VMDirectPath can be found in *Configuration Examples for VMDirectPath* [VV25] and *Configuring VMDirectPath* [VV26].

Appendix F Hard-disk space needed for the project

The following table details the hard-disk space needed for the project on **one** hardware platform:

Product	Partition	Needed space	Note
AIDA-NG	`/`	30 GB	Software
	`/boot`	100 MB	
	`/var`	30 GB	Database: 100 days x 300 MB/day
	Total	61 GB	
CADAS-ATS	`/`	30 GB	Software
	`/boot`	100 MB	
	`/var`	50 GB	Database
	Total	81 GB	
CADAS-IMS	`/`	30 GB	Software
	`/boot`	100 MB	
	`/shared`	50 GB	Database
	`/var`	5 GB	
	Total	86 GB	
Total		**228 GB**	

Table F-1: Hard-disk space needed by Comsoft products

With a RAID 5 group over three hard-disk drives of 140 GB, we theoretically obtain a total space of 280 GB. A few tests showed that the actual space is 270 GB.

ESXi software needs less than 5 GB, but free space is needed in case the total main memory assigned to virtual machines is bigger than the physical available main memory. ESXi would use this space as a swap space. Even if we are sure that ESXi will never need to swap memory pages out (see chapter 3.5.1.2 Main Memory), ESXi will refuse to start a virtual machine if there is not enough free space on the hard-disk. If we assign the whole available main memory (12 GB) to all three virtual machines, a free space of 36 GB is needed.

270 GB - 36 GB - 5 GB = **229 GB > 228 GB**

Conclusion: three hard-disk drives are sufficient for our project.

Appendix G Savings due to the project

Note: This is an approximation. Some aspects (energy, postage, documentation writing, training, etc.) couldn't be taken into account.

The web page http://egui.houston.hp.com/eGlue/eco/begin.do allows to calculate the price of a server. Without virtualization (see chapter 3.1 Existing design), six servers with the following configuration are usually purchased:

- Model: HP DL380G6 E5520;
- Number of processors: 1;
- RAM: 4 GB (2 x 2 GB) PC3-10600R;
- 1 PCI Express Card: HP NC364T 4Pt (Gigabit NIC);
- DVD Optical Kit;
- 3x HP 146GB 3G SAS 10K SFF DP ENT HDD;
- 1 Redundant Power Supply.

The price of such a server is 4 524 USD without service nor support, which makes a total of 27 144 USD.

For our project, only two servers are needed (see chapter 4 Implementation), with the same configuration except on following points:

- Number of processors: 2;
- RAM: 12 GB (6 x 2 GB) PC3-10600R;
- 2 PCI Express Cards: HP NC364T 4Pt (Gigabit NIC).

The price of such a server is 6 372 USD without service nor support, which makes a total of 12 744 USD. **The hardware costs for the servers have been divided by 2.1.** Moreover the energy consumption of the whole solution will be lower.

HP assistance contracts (Service and Support) are called Care Packs. One Care Pack concerns only one server, making the price of the assistance proportional to the number of servers. **Our project divides the assistance costs by 3.**

Appendix H Health of an AIDA-NG Recording Sub-System

Several indicators exist to know if an RSS suffers from high latencies.

rsdb<N>times.txt

Monitor the file `/var/aida/db/rsdb<N>times.txt`, N being the ID of the RSS:

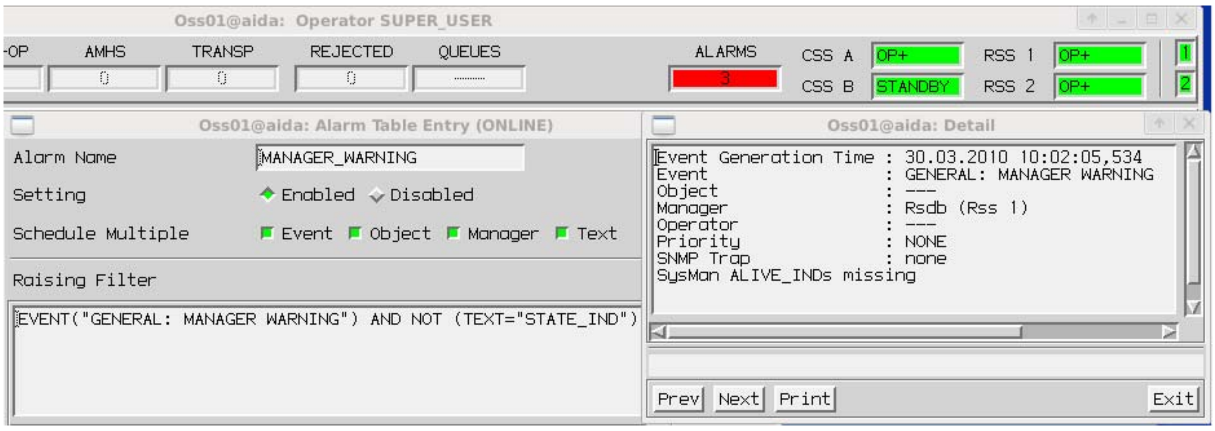
```
$ tail -f /var/aida/db/rsdb1times.txt
```

This file is a developer log-file. When an RSS performs an operation on its database, it measures the time needed to do it. Except when the RSS starts, such an operation should never last longer than 500 ms. Otherwise such a line is appended at the end of the file:

```
29.03.2010 17:37:00,743 : req getDiagData  reqSize 0 cnfSize 962 :
→   916[msec], # mbox msgs: 0
```

Manager Warning

When a RSS is having severe difficulties, it fails to regularly send *alive indicators* to its *system manager* (see chapter 2.1.1 Comsoft products), which triggers *manager warnings*, as shown in figure H-1. These warnings can be detected by means of an *alarm*.



Oss01@aida: Operator SUPER_USER

Oss01@aida: Alarm Table Entry (ONLINE)

```
Alarm Name          MANAGER_WARNING
Setting             ◆ Enabled  ◇ Disabled
Schedule Multiple   ✓ Event  ✓ Object  ✓ Manager  ✓ Text
Raising Filter
EVENT("GENERAL: MANAGER WARNING") AND NOT (TEXT="STATE_IND")
```

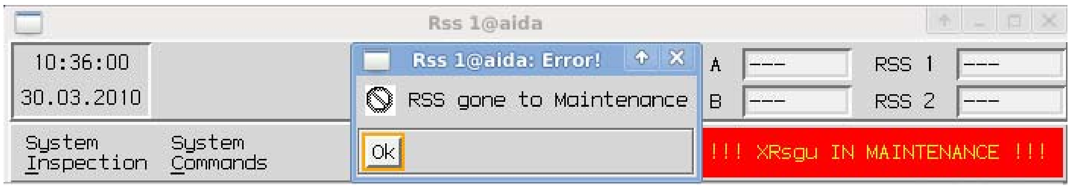
Oss01@aida: Detail

```
Event Generation Time : 30.03.2010 10:02:05,534
Event                 : GENERAL: MANAGER WARNING
Object                : ---
Manager               : Rsdb (Rss 1)
Operator              : ---
Priority              : NONE
SNMP Trap             : none
SysMan ALIVE_INDs missing
```

Figure H-1: Alive indicators missing

Maintenance

After five missing alive indicators, the RSS is switched off by its system manager (see chapter 2.1.1 Comsoft products). It is declared as being in maintenance:



Rss 1@aida: Error!

RSS gone to Maintenance

!!! XRsgu IN MAINTENANCE !!!

Figure H-2: RSS in maintenance

References

Aeronautics

[A1] Official website. *COMSOFT Corporate Website*. http://www.comsoft.aero/ (last accessed on 2010-04-22)

[A2] Official website section. EUROCONTROL – *Aeronautical Information Management*. http://www.eurocontrol.int/aim/public/subsite_homepage/homepage.html (last accessed on 2010-03-05)

[A3] EUROCONTROL. *Recommendations for ANS Software*. V1.0. 2005. Electronic document. http://www.dcs.gla.ac.uk/~johnson/Eurocontrol/software/SW_V1/ (last accessed on 2010-03-24). SAF.ET1.ST03.1000.GUI-01-00

[A4] Comsoft internal document. *AFTN/AMHS System – System Requirements Specification (SRS)*. V1.3, 2007-04-24. 37 p. AFTN/AMHS-SRS-V1.3

[A5] Comsoft. *AIDA-NG: Product Information*. 12 p. http://www.comsoft.aero/download/atc/product/english/aida_ng.pdf (last accessed on 2010-03-04)

[A6] Comsoft internal document. *AIDA-NG R09.2: Interface Control Document*. V1.3, 2009. 103 p. AIDA-NG-A-ICD-V1.3

[A7] Comsoft internal document. *AIDA-NG R09.2: Routing*. V1.4, 2009-01-22. 35 p. AIDA-NG-Routing-V1.4

[A8] Comsoft internal document. *AIDA-NG R09.2: System Architecture*. V1.0, 2009-08-31. 43 p. AIDA-NG-System-Architecture-V1.0

[A9] Comsoft. *CADAS-ATS: Product Information*. 4 p. http://www.comsoft.aero/download/atc/product/english/cadas_ats.pdf (last accessed on 2010-03-04)

[A10] Comsoft internal document. *CADAS-ATS: Administrator's Guide*. V1.3, 2009-03-11. 51 p. 3990_HEAD_CadasAtm_AdmGuide-V1.3

[A11] Comsoft. *CADAS-IMS: Product Information*. 6 p. http://www.comsoft.aero/download/atc/product/english/cadas_ims.pdf (last accessed on 2010-03-04)

[A12] Comsoft internal document. *CADAS-IMS: Technical Specification*. V1.2, 2009-04-15. 79 p. C-IMS-V1.2

[A13] Comsoft internal document. *CCMS R10.1: System Architecture*. V1.0, 2010-02-10. 20 p. CCMS-SysArc-R10.1-V1.0

[A14] Comsoft internal document. *CCMS: Administrator's Guide*. V1.5, 2010-03-12. 32 p. CCMS-AdmGuide-V1.5

[A15] Comsoft internal document. *CNMS: System Architecture*. V1.0, 2009-06-09. 21 p. CNMS-SysArch-V1.0

[A16] Comsoft internal document. *EFG: Administrator's Guide*. V1.8, 2010-02-18. 48 p. EFG-AdmGuide-V1.8

[A17] Comsoft internal document. *ATN Router Course*. Training presentation. 295 p. 2010-03-05

[A18] Comsoft internal document. *CADAS-AIM$_{DB}$ R3.2: Architectural Design Documentation*. V1.0, 2010-03-05. 11 p. CADAS-AIMDB-ADD-R3.2-V1.0

[A19] Comsoft internal document. *LAN Installation Guide*. V1.1, 2009-17-11. 32 p. LAN-InstGuide-V1.1

[A20] Comsoft internal document. *AIDA-NG R09.2: Reliability Testscript*. V2.0, 2009-09-18. 34 p. AIDA-NG-Reliability-V2.0

[A21] Comsoft internal document. *AIDA-NG R09.2: Performance Testscript*. V2.2, 2009-12-16. 8 p. AIDA-NG-Performance-V2.2

Computer architecture and operating systems

[C1] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture.* 2009. Electronic document. Order Number: 253665-032US

[C2] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2.* 2009. Electronic document. Order Number: 253669-032US

[C3] Intel. *Intel 80386 Programmer's Reference Manual*. 1986. 421 p. Electronic document

[C4] **Hennessy, John L.; Patterson, David A.** *Computer Architecture: A Quantitative Approach*. 4th edition. San Francisco, USA: Morgan Kaufmann Publishers, 2007. 423 p. 978-0-12-370490-0

[C5] **Brinkschulte, Uwe; Ungerer, Theo.** *Mikrocontroller und Mikroprozessoren*. 2nd edition. Berlin, Germany: Springer-Verlag, 2007. 453 p. 978-3-540-46801-1

[C6] **Ungerer, Theo.** *Parallelrechner und parallele Programmierung*. Berlin, Germany: Spektrum Akademischer Verlag, 1997. 375 p. 978-3827402318

[C7] **Stallings, William.** *Operating Systems: Internals and Design Principles*. 6th edition. Upper Saddle River, USA: Pearson Prentice Hall, 2009. 832 p. 978-0-13-600632-9

[C8] **Bovet, Daniel P.; Cesati, Marco.** *Understanding the Linux Kernel.* 3rd edition. Sebastopol, USA: O'Reilly Media, 2005. 942 p. 978-0-596-00565-8

[C9] **Love, R.** *Linux Kernel Development.* 2nd edition. Novell Press, 2005. 432 p. 978-0-672-32720-9

[C10] **Schroeder, Michael D.; Saltzer, Jerome H.** *A Hardware Architecture for Implementing Protection Rings.* New York, USA: ACM, 1972. Communications of the ACM 15(3), 157-170. ISSN 0163-5980. DOI= http://doi.acm.org/10.1145/850614.850623

[C11] HP. *HP ProLiant DL380 Generation 6, Worldwide QuickSpecs*. Version 10. 2009. Electronic document. DA – 13234

[C12] HP. *HP ProLiant RAID Smart Array controllers*. Electronic document. http://h18000.www1.hp.com/products/servers/proliantstorage/arraycontrollers/ (last accessed on 2010-02-18)

[C13] HP. *HP SmartArray P410 Controller*. Version 9. 2010. Electronic document. DA – 13201

[C14] Intel. *Intel Xeon Processor E5520*. Electronic document. http://ark.intel.com/Product.aspx?id=40200 (last accessed on 2010-01-04)

[C15] Intel. *Intel Xeon Processor 5500 Series, Datasheet, Volume 1*. 2009. Electronic document. Document Number: 321321-001

[C16] Intel. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*. 2008. Electronic document. http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf (last accessed on 2010-01-20)

[C17] Official website. *Fedora Project*. http://fedoraproject.org/ (last accessed on 2010-01-04)

[C18] Fedora. *Fedora 12 Release Notes*. 2009. Electronic document.

[C19] *8253 functions (General overview).* 2002. Electronic document. http://www.sharpmz.org/mz-700/8253ovview.htm (last accessed on 2010-01-19)

[C20] **Friesen, Brandon.** *Kernel development tutorial*. 1999. Electronic document. http://www.osdever.net/bkerndev/ (last accessed on 2010-01-19)

[C21] The Linux Foundation. *Bonding*. 2009. Electronic document. http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding (last accessed on 2010-03-08)

[C22] Official website. *LMBench*. http://www.bitmover.com/lmbench/ (last accessed on 2010-03-09)

[C23] **Staelin, Carl.** *lmbench – an extensible micro-benchmark suite*. Great Britain: Software Practice and Experience, 2005. Vol. 35; Numb. 11, 1079-1105. ISSN 0038-0644

[C24] **Peng, Lu; Peir, Jih-Kwon; Prakash, Tribuvan K.; Staelin, Carl; Chen, Yen-Kuang; Koppelman, David.** *Memory hierarchy performance measurement of commercial dual-core desktop processors*. Journal of Systems Architecture 54 (2008) 816–828. ISSN 1383-7621

[C25] **Maxwell, M. Tyler; Cameron; Kirk W.** *Optimizing Application Performance: A Case Study Using LMBench*. Crossroads, the ACM Student Magazine, 2002. Electronic document. http://www.acm.org/crossroads/xrds8-5/optaperf.html (available 2010-03-09)

[C26] **Jaenisch, Volker; Klapproth, Martin; Westphal, Patrick.** *I/O-Scheduler und RAID-Performance*. Version 5. Linux Technical Review. 16 p. Electronic document. http://www.linuxtechnicalreview.de/Themen/Performance-and-Tuning/I-O-Scheduler-und-RAID-Performance (last accessed on 2010-03-11)

Real-Time

[CR1] **Wörn, Heinz; Brinkschulte, Uwe.** *Echtzeitsysteme*. Berlin, Germany: Springer-Verlag, 2005. 556 p. 978-3-540-20588-3

[CR2] Official website. *Wind River: RTLinuxFree*. http://www.rtlinuxfree.com/ (last accessed on 2009-12-18)

[CR3] **List, Stéphane; Ferre, Nicolas.** *RTLinux.* V1.22. Saint-Denis, France: Alcôve. 44 p. Electronic document. http://nferre.free.fr/rtl/rtlinux_sl_v3_r.pdf (last accessed on 2010-01-18)

[CR4] **Rivers, Brian.** *How to install RTLinux 3.1 with RedHat 7.1, 7.2 & 7.3*. Electronic document. http://www.brivers.net/rtlinux-install-howto.html (last accessed on 2010-01-18)

[CR5] **Divakaran, Dinil.** *RTLinux HOWTO.* V1.1. Electronic document. http://tldp.org/HOWTO/RTLinux-HOWTO.html (last accessed on 2010-01-18)

[CR6] **Ficheux, Pierre; Kadionik, Patrice.** *Temps réel sous Linux*. 2003. 37 p. Electronic document. http://uuu.enseirb.fr/~kadionik/embedded/linux_realtime/linux_realtime.pdf (last accessed on 2010-01-18)

[CR7] **McGuire, Nicholas.** *MiniRTL-V2.3 (Kernel 2.2.14).* Electronic document. http://www.rtlinux-gpl.org/cgi-bin/viewcvs.cgi/rtldoc-3.2-pre1/doc/html/minirtl.html?rev=1.1.1.1 (last accessed on 2010-01-18)

[CR8] Official website. *Xenomai*. http://www.xenomai.org/ (last accessed on 2010-01-18)

[CR9] **Cuvillon, Loïc.** *Systèmes temps réel et systèmes embarqués*. 2008. 53 p. Electronic document. http://eavr.u-strasbg.fr/wiki/upload/8/81/TR_chap_linux_temps_reel_08_poly.pdf (last accessed on 2010-01-18)

[CR10] Official website. *RTAI*. http://www.rtai.org/ (last accessed on 2010-01-18)

[CR11] **Santana, Pedro Henrique; Amui, Bruno Guilherme; Cavalcanti, Felipe Brandão; Scandaroli, Glauco Garcia; Borges, Geovany Araújo.** *Building a real-time Debian distribution for embedded systems.* 2010. 11 p. Electronic document. http://www.lara.ene.unb.br/~phsantana/files/technotes/HowTo_Debian_Embedded.pdf (last accessed on 2010-01-18)

[CR12] eAVR. *Systèmes temps-réel et systèmes embarqués*. Electronic document. http://eavr.u-strasbg.fr/wiki/index.php/Syst%C3%A8mes_temps-r%C3%A9el_et_syst%C3%A8mes_embarqu%C3%A9s (last accessed on 2010-01-18)

[CR13] Xenomai. *Xenomai API Documentation*. Electronic document. http://www.xenomai.org/documentation/xenomai-head/html/api/ (last accessed on 2010-01-18)

[CR14] Official website. *The FreeDOS Project*. http://www.freedos.org/ (last accessed on 2010-01-18)

[CR15] Official website section. *DJGPP*. http://www.delorie.com/djgpp/ (last accessed on 2010-01-18)

High Availability

[H1] Linux-Magazin. *High Availability.* 4th edition. Munich, Germany: Linux New Media AG, 2007. 144 p. 978-3-939551-06-5

Telecommunication

[N1] **Dooley, Kevin.** *Designing Large-Scale LANs.* Sebastopol, USA: O'Reilly Media, 2002. 400 p. 978-0-596-00150-6

[N2] Official website. *View500 Directory Server*. http://www.view500.com/ (last accessed on 2010-03-05)

[N3] Cisco Systems. *Cisco Catalyst 3560 Series Switches: Data Sheet*. 2009. 21 p. Doc.-ID: C78-379068-08

[N4] Cisco Systems. *Catalyst 3560 Switch Software Configuration Guide*. Cisco IOS Release 12.2(52)SE. San Jose, USA: 2009. 1296 p. Doc.-ID: OL-8553-07

Virtualization

[V1] **Thorns, Fabian.** *Das Virtualisierungs-Buch*. 1st edition. Böblingen, Germany: Computer und Literaturverlag, 2007. 665 p. 978-3-936546-43-9

[V2] **Popek, Gerald J.; Goldberg, Robert P.** *Formal requirements for virtualizable third generation architectures.* New York, USA: ACM, 1974. Communications of the ACM 17(7), 412-421. ISSN 0001-0782. DOI= http://doi.acm.org/10.1145/361011.361073

[V3] **Scott Robin, John; Irvine, Cynthia E.** *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor.* Denver, USA, 2000. Proceedings of the 9th USENIX Security Symposium

[V4] **Lawton, Kevin.** *Running multiple operating systems concurrently on an IA32 PC using virtualization techniques*. 1999. Electronic document. http://www.ece.cmu.edu/~ece845/sp05/docs/plex86.txt (last accessed on 2009-12-08)

[V5] **Neiger, Gil; Santoni, Amy; Leung, Felix; Rodgers, Dion; Uhlig, Rich.** *Intel® Virtualization Technology: Hardware support for efficient processor virtualization.* Intel Technology Journal, Volume 10, Issue 03. 2006-08-10. ISSN 1535-864X. DOI= 10.1535/itj.1003.01

[V6] **Adams, Keith; Alesen, Ole.** *A Comparison of Software and Hardware Techniques for x86 Virtualization.* New York, USA: ACM, 2006. ASPLOS-XII, 2-13, DOI= http://doi.acm.org/10.1145/1168857.1168860

[V7] HP Laboratories. **Padala, Pradeep; Zhu, Xiaoyun; Wang, Zhikui; Singhal, Sharad; Shin, Kang G.** *Performance Evaluation of Virtualization Technologies for Server Consolidation.* 2007. HPL-2007-59R1

[V8] Red Hat. **Curran, Christopher; Holzer, Jan Mark.** *Red Hat Enterprise Linux 5: Virtualization Guide*. 4th edition. 2009

[V9] **Burger, Thomas.** *The Advantages of Using Virtualization Technology in the Enterprise.* 2008. Electronic document. http://software.intel.com/en-us/articles/the-advantages-of-using-virtualization-technology-in-the-enterprise/ (last accessed on 2009-12-11)

[V10] **Daniels, Jeff.** *Server Virtualization Architecture and Implementation*. ACM Crossroads, Fall 2009 / Vol. 16, No. 1

[V11] **Mergen, Mark F.; Uhlig, Volkmar; Krieger, Orran; Xenidis Jimi.** *Virtualization for High-Performance Computing*. SIGOPS Oper. Syst. Rev. 40, 2 (Apr. 2006), 8-11. DOI= http://doi.acm.org/10.1145/1131322.1131328

[V12] **Fong, Liana; Steinder, Malgorzata.** *Duality of Virtualization: Simplification and Complexity.* SIGOPS Oper. Syst. Rev. 42, 1 (Jan. 2008), 96-97. DOI= http://doi.acm.org/10.1145/1341312.1341330

[V13] **Childers, Bill.** *Virtualization Shootout: VMware Server vs. VirtualBox vs. KVM*. Linux J. 2009, 187 (Nov. 2009), 12

[V14] **Li, Peng.** *Selecting and using virtualization solutions: our experiences with VMware and VirtualBox*. J. Comput. Small Coll. 25, 3 (Jan. 2010), 11-17

[V15] **Miller, Karissa; Pegah, Mahmoud.** *Virtualization: virtually at the desktop*. In Proceedings of the 35th Annual ACM SIGUCCS Conference on User Services (Orlando, Florida, USA, October 07 - 10, 2007). SIGUCCS '07. ACM, New York, NY, 255-260. DOI= http://doi.acm.org/10.1145/1294046.1294107

[V16] **Crosby, Simon; Brown, David.** *The Virtualization Reality.* Queue 4, 10 (Dec. 2006), 34-41. DOI= http://doi.acm.org/10.1145/1189276.1189289

[V17] **Apparao, Padma; Iyer, Ravi; Zhang, Xiaomin; Newell, Don; Adelmeyer, Tom.** *Characterization & Analysis of a Server Consolidation Benchmark*. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Seattle, WA, USA, March 05 - 07, 2008). VEE '08. ACM, New York, NY, 21-30.
DOI= http://doi.acm.org/10.1145/1346256.1346260

[V18] **Matthews, Jeanna Neefe; Hu, Wenjin; Hapuarachchi, Madhujith; Deshane, Todd; Dimatos, Demetrios; Hamilton, Gary; McCabe, Michael; Owens, James.** *Quantifying the Performance Isolation Properties of Virtualization Systems.* In Proceedings of the 2007 Workshop on Experimental Computer Science (San Diego, California, June 13 - 14, 2007). ExpCS '07. ACM, New York, NY, 6. DOI= http://doi.acm.org/10.1145/1281700.1281706

[V19] **Tanaka, Tsuyoshi; Tarui, Toshiaki; Naono, Ken.** *Investigating Suitability for Server Virtualization using Business Application Benchmarks*. In Proceedings of the 3rd international Workshop on Virtualization Technologies in Distributed Computing (Barcelona, Spain, June 15 - 15, 2009). VTDC '09. ACM, New York, NY, 43-50. DOI= http://doi.acm.org/10.1145/1555336.1555344

[V20] **Soltesz, Stephen; Pötzl, Herbert; Fiuczynski, Marc E.; Bavier, Andy; Peterson, Larry.** *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*. SIGOPS Oper. Syst. Rev. 41, 3 (Jun. 2007), 275-287. DOI= http://doi.acm.org/10.1145/1272998.1273025

[V21] **Drepper, Ulrich.** *The Cost of Virtualization*. 2008. Queue 6, 1 (Jan. 2008), 28-35. DOI= http://doi.acm.org/10.1145/1348583.1348591

[V22] **Kaiser, Robert.** *Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems.* In Proceedings of the 1st Workshop on Isolation and integration in Embedded Systems (Glasgow, Scotland, April 01 - 01, 2008). M. Engel and O. Spinczyk, Eds. IIES '08. ACM, New York, NY, 5-10.
DOI= http://doi.acm.org/10.1145/1435458.1435460

[V23] **Chadha, Vineet; Figueiredo, Renato J.; Illikkal, Ramesh; Iyer, Ravi; Moses, Jaideep; Newell, Donald.** *I/O Processing in a Virtualized Platform: A Simulation-Driven Approach*. In Proceedings of the 3rd international Conference on Virtual Execution Environments (San Diego, California, USA, June 13 - 15, 2007). VEE '07. ACM, New York, NY, 116-125.
DOI= http://doi.acm.org/10.1145/1254810.1254827

[V24] **Dong, Yaozu; Dai, Jinquan; Huang, Zhiteng; Guan, Haibing; Tian, Kevin; Jiang, Yunhong.** *Towards High-Quality I/O Virtualization*. In Proceedings of SYSTOR 2009: the Israeli Experimental Systems Conference (Haifa, Israel). SYSTOR '09. ACM, New York, NY, 1-8.
DOI= http://doi.acm.org/10.1145/1534530.1534547

[V25] **Kumar Ram, Kaushik; Santos, Jose Renato; Turner, Yoshio; Cox, Alan L.; Rixner, Scott.** *Achieving 10 Gb/s using Safe and Transparent Network Interface Virtualization*. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Washington, DC, USA, March 11 - 13, 2009). VEE '09. ACM, New York, NY, 61-70.
DOI= http://doi.acm.org/10.1145/1508293.1508303

[V26] **Kim, Hwanju; Lim, Hyeontaek; Jeong, Jinkyu; Jo, Heeseung; Lee, Joowon.** *Task-aware Virtual Machine Scheduling for I/O Performance*. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Washington, DC, USA, March 11 - 13, 2009). VEE '09. ACM, New York, NY, 101-110. DOI= http://doi.acm.org/10.1145/1508293.1508308

[V27] **Weng, Chuliang; Wang, Zhigang; Li, Minglu; Lu, Xinda.** *The Hybrid Scheduling Framework for Virtual Machine Systems*. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Washington, DC, USA, March 11 - 13, 2009). VEE '09. ACM, New York, NY, 111-120. DOI= http://doi.acm.org/10.1145/1508293.1508309

[V28] **Liao, Guangdeng; Guo, Danhua; Bhuyan, Laxmi; King, Steve R.** *Software Techniques to Improve Virtualized I/O Performance on Multi-Core Systems*. In Proceedings of the 4th ACM/IEEE Symposium on Architectures For Networking and Communications Systems (San Jose, California, November 06 - 07, 2008). ANCS '08. ACM, New York, NY, 161-170.
DOI= http://doi.acm.org/10.1145/1477942.1477971

[V29] **Liu, Jiuxing; Abali, Bulent.** *Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization*. In Proceedings of the 23rd international Conference on Supercomputing (Yorktown Heights, NY, USA, June 08 - 12, 2009). ICS '09. ACM, New York, NY, 225-234.
DOI= http://doi.acm.org/10.1145/1542275.1542309

[V30] **Raj, Himanshu; Schwan, Karsten.** *High Performance and Scalable I/O Virtualization via Self-Virtualized Devices.* In Proceedings of the 16th international Symposium on High Performance Distributed Computing (Monterey, California, USA, June 25 - 29, 2007). HPDC '07. ACM, New York, NY, 179-188.
DOI= http://doi.acm.org/10.1145/1272366.1272390

[V31] **Wells, Philip M.; Chakraborty, Koushik; Sohi, Gurindar S.** *Dynamic Heterogeneity and the Need for Multicore Virtualization.* SIGOPS Oper. Syst. Rev. 43, 2 (Apr. 2009), 5-14. DOI= http://doi.acm.org/10.1145/1531793.1531797

[V32] **Huang, Lan; Peng, Gang; Chiueh, Tzi-cker.** *Multi-Dimensional Storage Virtualization.* SIGMETRICS Perform. Eval. Rev. 32, 1 (Jun. 2004), 14-24.
DOI= http://doi.acm.org/10.1145/1012888.1005692

[V33] **Ongaro, Diego; Cox, Alan L.; Rixner, Scott.** *Scheduling I/O in Virtual Machine Monitors*. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Seattle, WA, USA, March 05 - 07, 2008). VEE '08. ACM, New York, NY, 1-10.
DOI= http://doi.acm.org/10.1145/1346256.1346258

[V34] **Gernot Heiser.** *The Role of Virtualization in Embedded Systems.* In Proceedings of the 1st Workshop on Isolation and integration in Embedded Systems (Glasgow, Scotland, April 01 - 01, 2008). M. Engel and O. Spinczyk, Eds. IIES '08. ACM, New York, NY, 11-16.
DOI= http://doi.acm.org/10.1145/1435458.1435461

[V35] **Seelam, Seetharami R.; Teller, Patricia J.** *Virtual I/O Scheduler: A Scheduler of Schedulers for Performance Virtualization*. In Proceedings of the 3rd international Conference on Virtual Execution Environments (San Diego, California, USA, June 13 - 15, 2007). VEE '07. ACM, New York, NY, 105-115.
DOI= http://doi.acm.org/10.1145/1254810.1254826

[V36] **Rusty Russell.** *virtio: Towards a De-Facto Standard For Virtual I/O Devices*. SIGOPS Oper. Syst. Rev. 42, 5 (Jul. 2008), 95-103.
DOI= http://doi.acm.org/10.1145/1400097.1400108

[V37] **Zhang, Jianyong; Sivasubramaniam, Anand; Wang, Qian; Riska, Alma; Riedel, Erik.** *Storage Performance Virtualization via Throughput and Latency Control.* Trans. Storage 2, 3 (Aug. 2006), 283-308.
DOI= http://doi.acm.org/10.1145/1168910.1168913

[V38] **Ray, Edward; Schultz, Eugene.** *Virtualization Security.* In Proceedings of the 5th Annual Workshop on Cyber Security and information intelligence Research: Cyber Security and information intelligence Challenges and Strategies (Oak Ridge, Tennessee, April 13 - 15, 2009). F. Sheldon, G. Peterson, A. Krings, R. Abercrombie, and A. Mili, Eds. CSIIRW '09. ACM, New York, NY, 1-5.
DOI= http://doi.acm.org/10.1145/1558607.1558655

[V39] **Burdonov, Igor; Kosachev, Alexander; Iakovenko, Pavel.** *Virtualization-based separation of privilege: working with sensitive data in untrusted environment.* In Proceedings of the 1st Eurosys Workshop on Virtualization Technology For Dependable Systems (Nuremberg, Germany, March 31 - 31, 2009). VDTS '09. ACM, New York, NY, 1-6. DOI= http://doi.acm.org/10.1145/1518684.1518685

[V40] **Huang, Yih; Stavrou, Angelos; Ghosh, Anup K.; Jajodia, Sushil.** *Efficiently Tracking Application Interactions using Lightweight Virtualization.* In Proceedings of the 1st ACM Workshop on Virtual Machine Security (Alexandria, Virginia, USA, October 27 - 27, 2008). VMSec '08. ACM, New York, NY, 19-28.
DOI= http://doi.acm.org/10.1145/1456482.1456486

[V41] **Sharif, Monirul; Lee, Wenke; Cui, Weidong; Lanzi, Andrea.** *Secure In-VM Monitoring Using Hardware Virtualization.* In Proceedings of the 16th ACM Conference on Computer and Communications Security (Chicago, Illinois, USA, November 09 - 13, 2009). CCS '09. ACM, New York, NY, 477-487.
DOI= http://doi.acm.org/10.1145/1653662.1653720

[V42] **Chen, Haibo; Chen, Rong; Zhang, Fengzhe; Zang, Binyu; Yew, Pen-Chung.** *Live Updating Operating Systems Using Virtualization.* In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM, New York, NY, 35-44.
DOI= http://doi.acm.org/10.1145/1134760.1134767

[V43] **Laarouchi, Youssef; Deswarte, Yves; Powell, David; Arlat, Jean; De Nadai, Eric.** *Enhancing Dependability in Avionics Using Virtualization.* In Proceedings of the 1st Eurosys Workshop on Virtualization Technology For Dependable Systems (Nuremberg, Germany, March 31 - 31, 2009). VDTS '09. ACM, New York, NY, 13-17. DOI= http://doi.acm.org/10.1145/1518684.1518687

[V44] Official website. *WineHQ*. http://www.winehq.org/ (last accessed on 2009-12-11)

[V45] Official website. **Biallas, Sebastian.** *PearPC.* http://pearpc.sourceforge.net/ (last accessed on 2009-12-04)

[V46] Official website. *VirtualLogix*. http://www.virtuallogix.com/ (last accessed on 2010-03-04)

[V47] Official website. *Real-Time Systems*. http://www.real-time-systems.com/ (last accessed on 2010-03-04)

[V48] KUKA RTOS. *Virtualizing Real-Time Operating Systems with Windows*. Electronic document. http://virtualization.kuka-rtos.com/ (last accessed on 2010-03-04)

[V49] National Instruments. *NI Real-Time Hypervisor Architecture and Performance Details*. 2009. Electronic document. http://zone.ni.com/devzone/cda/tut/p/id/9629 (last accessed on 2010-03-04)

[V50] Tutorial. *Introduction to Dynamic Recompilation.* Electronic document. http://web.archive.org/web/20051018182930/www.zenogais.net/Projects/Tutorials/Dynamic+Recompiler.html (last accessed on 2009-12-08)

[V51] Tutorial. **Toal, Graham.** *An Emulator Writer's HOWTO for Static Binary Translation*. http://www.gtoal.com/sbt/ (last accessed on 2009-12-09)

KVM

[VK1] Fedora. Getting started with virtualization. Electronic document. http://fedoraproject.org/wiki/Virtualization_Quick_Start (last accessed on 2010-01-26)

[VK2] **Habib, Irfan.** *Virtualization with KVM*. *Linux J.* 2008, 166 (Feb. 2008), 8

[VK3] Official website. *KVM*. http://www.linux-kvm.org/ (last accessed on 2010-01-28)

[VK4] Official website. *QEMU*. http://wiki.qemu.org/ (last accessed on 2010-03-03)

[VK5] Official website. *The virtualization API*. http://libvirt.org/ (last accessed on 2010-01-28)

[VK6] **Shah, Amit.** *Kernel-based virtualization with KVM*. In Linux Magazine, Issue #86, January 2008.

[VK7] KVM. *How to assign devices with VT-d in KVM*. Electronic document. http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM (last accessed on 2010-03-03)

Microsoft

[VM1] Microsoft. *Microsoft Virtual Server 2005 R2 SP1 – Enterprise Edition*. Electronic document. http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=bc49c7c8-4840-4e67-8dc4-1e6e218acce4 (last accessed on 2010-03-04)

[VM2] Microsoft. *Virtual Server 2005 R2 SP1 Download Frequently Asked Questions*. 2007. Electronic document. http://technet.microsoft.com/en-us/virtualserver/bb676680.aspx (last accessed on 2010-03-04)

[VM3] **Howard, John.** *Virtual Server 2005 R2 SP1 Beta 1 download link and availability details*. 2006. Electronic document. http://blogs.technet.com/jhoward/archive/2006/04/28/426703.aspx (last accessed on 2010-03-04)

Linux VServer

[VS1] Official website. *Linux-VServer*. http://linux-vserver.org/ (last accessed on 2010-03-04)

[VS2] **Campbell, C.** *Installation on Fedora*. Electronic document. http://linux-vserver.org/Installation_on_Fedora (last accessed on 2010-03-04)

VMware

[VV1] VMware. *VMware Company Overview*. 2009. Electronic document. http://www.vmware.com/files/pdf/VMware-Company-Overview-DS-EN.pdf (last accessed on 2010-03-02)

[VV2] VMware. *VMware Infrastructure*. Electronic document. http://www.vmware.com/products/vi/overview.html (last accessed on 2010-02-22)

[VV3] VMware. *VMware ESX and VMware ESXi*. Electronic document. http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf (last accessed on 2010-02-28)

[VV4] VMware. *VMware Management Assistant Guide. (vSphere 4.0).* 34 p. Electronic document. EN-000116-00

[VV5] VMware. *VMware Virtual SMP (product datasheet).* Electronic document. http://www.vmware.com/pdf/vsmp_datasheet.pdf (last accessed on 2010-03-02)

[VV6] VMware. *VMware Server 2: A Risk-Free Way to Get Started with Virtualization*. 2009. Electronic document. http://www.vmware.com/files/pdf/VMware-Server-2-DS-EN.pdf (last accessed on 2010-03-02)

[VV7] **Troy, Ryan; Helmke, Matthew.** *VMware Cookbook*. O'Reilly Media, 2009. 304 p. 978-0-596-15725-8

[VV8] **Larisch, Dirk.** *Praxisbuch VMware Server – Das praxisorientierte Nachschlagewerk zu VMware Server.* Wuppertal, Germany: Carl Hanser Verlag München Wien, 2007. 498 p. 978-3-446-40901-9

[VV9] VMware. *VMware Workstation 7: The Gold Standard in Desktop Virtualization*. 2009. Electronic document. http://www.vmware.com/files/pdf/VMware-Workstation-7-DS-EN.pdf (last accessed on 2010-03-02)

[VV10] VMware. *Timekeeping in VMware Virtual Machines.* 2008. Electronic document. http://www.vmware.com/pdf/vmware_timekeeping.pdf (last accessed on 2009-12-17)

[VV11] VMware. *VMware ESXi 4.0 Update 1 Release Notes.* 2009. Electronic document. http://www.vmware.com/support/vsphere4/doc/vsp_esxi40_u1_rel_notes.html (last accessed on 2010-01-04)

[VV12] VMware. *Performance of VMware VMI*. 2008. Electronic document. http://www.vmware.com/pdf/VMware_VMI_performance.pdf (last accessed on 2010-01-05)

[VV13] VMware. *Performance Evaluation of Intel EPT Hardware Assist*. 2009. Electronic document. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf (last accessed on 2010-01-14)

[VV14] VMware. *Performance Evaluation of AMD RVI Hardware Assist*. 2009. Electronic document. http://www.vmware.com/pdf/RVI_performance.pdf (last accessed on 2010-01-14)

[VV15] VMware. *VMware Compatibility Guide*. Electronic document. http://www.vmware.com/resources/compatibility/search.php (last accessed on 2010-01-05)

[VV16] VMware. *Enabling Virtual Machine Interface (VMI) in a Linux kernel and in ESX 3.5*. 2009. Electronic document. http://kb.vmware.com/kb/1003644 (last accessed on 2010-01-14)

[VV17] VMware. *Update: Support for guest OS paravirtualization using VMware VMI to be retired from new products in 2010-2011*. 2009. Electronic document. http://blogs.vmware.com/guestosguide/2009/09/vmi-retirement.html (last accessed on 2010-01-14)

[VV18] VMware. *Migrating VMI-enabled virtual machines to platforms that do not support VMI*. 2009. Electronic document. http://kb.vmware.com/kb/1013842 (last accessed on 2010-01-14)

[VV19] VMware. *vSphere Basic System Administration (Update 1, ESX 4.0, ESXi 4.0, vCenter Server 4.0).* 2009. 368 p. Electronic document. EN-000260-00

[VV20] VMware. *Using esxtop to troubleshoot performance problems*. 2004. 7 p. Electronic document. ESX-PFP-Q404-003

[VV21] VMware. *VMware vSphere 4: The CPU Scheduler in VMware ESX 4*. 2009. 21 p. Electronic document. http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf (last accessed on 2010-01-24)

[VV22] Official website section. *Installing VMware Tools*. http://www.vmware.com/support/ws55/doc/new_guest_tools_ws.html (last accessed on 2009-12-09)

[VV23] VMware. *VMware Tools for Linux Guests*. Electronic document. http://www.vmware.com/support/ws55/doc/ws_newguest_tools_linux.html (last accessed on 2010-02-11)

[VV24] VMware. *Configuring Disks to use VMware Paravirtual SCSI adapters*. Electronic document. http://kb.vmware.com/kb/1010398 (last accessed on 2010-02-11)

[VV25] VMware. *Configuration Examples and Troubleshooting for VMDirectPath*. 2009. 5 p. Electronic document. EN-000217-00

[VV26] VMware. *Configuring VMDirectPath I/O pass-through devices on an ESX host*. Electronic document. http://kb.vmware.com/kb/1010789 (last accessed on 2010-02-18)

[VV27] VMware. *Choosing a network adapter for your virtual machine*. Electronic document. http://kb.vmware.com/kb/1001805 (last accessed on 2010-03-15)

[VV28] **Petri, Daniel.** *Traffic Shaping With VMware ESX Server*. 2009. Electronic document. http://www.petri.co.il/traffic_shaping_with_vmware_esx_server.htm (last accessed on 2010-03-16)

[VV29] HP. *HP and VMware: Virtualization to consolidate server resources for maximal efficiency*. 2003. Electronic document. 5982-0616EN

[VV30] VMware. *VMware and Intel Capital Announce Investment*. 2007. Electronic document. http://www.vmware.com/company/news/releases/vmware_intel.html (last accessed on 2010-03-02)

Xen

[VX1] Official website section. *Xen Hypervisor*. http://xen.org/products/xenhyp.html (last accessed on 2010-03-03)

[VX2] **Barham, Paul; Dragovic, Boris; Fraser, Keir; Hand, Steven; Harris, Tim; Ho, Alex; Neugebauer, Rolf; Pratt, Ian; Warfield Andrew.** *Xen and the Art of Virtualization*. New York, USA: ACM, 2003. Proceedings of the nineteenth ACM symposium on Operating systems principles, 164-177. 1-58113-757-5

[VX3] **Radonic, Andrej; Meyer, Frank.** *Xen 3*. Poing, Germany: Franzis Verlag, 2006. 439 p. 978-3-7723-7899-7

[VX4] XenSource. *Xen User's Manual: Xen v3.0*. 2005. Electronic document. http://tx.downloads.xensource.com/downloads/docs/user/user.html (last accessed on 2010-03-03)

[VX5] **Magenheimer, Dan.** *Memory Overcommit… without the commitment.* 2008. Xen Summit 2008. Electronic document. http://wiki.xensource.com/xenwiki/Open_Topics_For_Discussion?action=AttachFile&do=get&target=Memory+Overcommit.pdf (last accessed on 2009-12-10)

[VX6] **Dong, Yaozu; Li, Shaofan; Mallick, Asit; Nakajima, Jun; Tian, Kun; Xu, Xuefei; Yang, Fred; Yu, Wilfred.** *Extending Xen with Intel Virtualization Technology*. 2006. ISSN 1535-864X

[VX7] **Mathai, Jacob.** *Xen Scheduling*. 2007. Electronic document. http://wiki.xensource.com/xenwiki/Scheduling (last accessed on 2010-03-03)

[VX8] XenSource. *Xen FAQ*. 2010. Electronic document. http://wiki.xensource.com/xenwiki/XenFaq (last accessed on 2010-03-03)

[VX9] **Xu, Herbert.** *TCP/Generic Segmentation Offload and its application in Xen*. Electronic document. http://www.xen.org/files/summit_3/rdd-tso-xen.pdf (last accessed on 2010-03-03)

[VX10] **Perilli, Alessandro.** *Red Hat adopts KVM: what happens to Xen now?* 2008. Electronic document. http://www.virtualization.info/2008/06/red-hat-adopts-kvm-what-happens-to-xen.html (last accessed on 2010-03-03)

OpenVZ

[VZ1] **Kolyshkin, Kirill.** *Virtualization in Linux*. 2006. Electronic document. http://download.openvz.org/doc/openvz-intro.pdf (last accessed on 2009-12-10)

[VZ2] SWsoft, Inc. *OpenVZ User's Guide*. Version 2.7.0-8. 2005. Electronic document. http://download.openvz.org/doc/OpenVZ-Users-Guide.pdf (last accessed on 2009-12-10)

[VZ3] Official website. *OpenVZ Wiki*. http://wiki.openvz.org/ (last accessed on 2010-03-04)

[VZ4] OpenVZ. *Features*. 2010. Electronic document. http://wiki.openvz.org/Features (last accessed on 2010-03-04)

Glossary

ACPI	Advanced Configuration and Power Interface
AIDA-NG	Aeronautical Integrated Data Exchange Agent – Next Generation
AFTN	Aeronautical Fixed Telecommunication Network
AIM	Aeronautical Information Management
AMHS	ATS Message Handling System
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
AS	Application Server
ASM	Air Space Management
ATC	Air Traffic Control
ATFM	Air Traffic Flow Management
ATM	Air Traffic Management
ATN	Aeronautical Telecommunication Network
ATS	Air Traffic Services
BCET	Best Case Execution Time
CADAS	Comsoft's Aeronautical Data Access System
CCMS	Comsoft Configuration Management Suite
CFQ	Completely Fair Queuing
CLNP	Connectionless Network Protocol
CMOS	Complementary Metal Oxide Semiconductor
CNMS	Comsoft Network Management System
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSS	Core Sub-System
DB	Database
DMA	Direct Memory Access
DPM	Distributed Power Management (VMware)
DRS	Distributed Resource Scheduler (VMware)
EFG	E-Mail/Fax Gateway
ELAN	External LAN
EPT	Extended Page Table
FPL	Flight Plan

HDD	Hard Disk Drive
HTTP	Hypertext Transfer Protocol
ICAO	International Civil Aviation Organization
IDE	Integrated Drive Electronics
ILAN	Internal LAN
IMS	Information Management & Services
I/O	Input/Output
IP	Internet Protocol
iSCSI	Internet SCSI
KVM	Kernel-based Virtual Machine
LA	Logical Address
LAN	Local Area Network
MH	Message Handler
MMU	Memory Management Unit
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
NIC	Network Interface Card
NOTAM	Notification To Airmen
NTP	Network Time Protocol
NUMA	Non Uniform Memory Access
OPMET	Operational Meteorological
OS	Operating System
OSS	Operating Sub-System
OWP	Operator Working Position
PAE	Physical Address Extension
PCI	Peripheral Component Interconnect
PIT	Programmable Interval Timer
PML	Pending Message List
PXE	Pre-boot Execution Environment
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RPC	Remote Procedure Call
RPM	RPM Package Manager
RSS	Recording Sub-System

RTC	Real Time Clock
RVI	Rapid Virtualization Indexing (AMD)
SAS	Serial Attached SCSI
SCSI	Small Computer System Interface
SMP	Symmetric Multiprocessing
SNL	Serial Network Link
SPOF	Single Point of Failure
SRS	System Requirements Specification
SWAL	Software Assurance Level
TCP	Transmission Control Protocol
TLB	Translation Look-aside Buffer
TS	Terminal Server
TSC	Time Stamp Counter
TSO	TCP Segmentation Offloading
UA	User Agent
UDP	User Datagram Protocol
UPS	Uninterrupted Power Supply
VE	Virtual Environment
VLAN	Virtual LAN
VM	Virtual Machine
VMFS	Virtual Machine File System
VMI	Virtual Machine Interface
VMM	Virtual Machine Manager
VMX	Virtual Machine Extension
VNC	Virtual Network Computing
VPS	Virtual Private Server
VT	Virtualization Technology (Intel)
VT-d	Virtualization Technology for Directed I/O (Intel)
VT-x	Virtualization Technology on the x86 platform (Intel)
WAN	Wide Area Network
WCET	Worst Case Execution Time
WS	Web Server

End of document