CRANFIELD UNIVERSITY

Csaba Kerti

Virtual Machine Co-Scheduling Modules

School of Engineering
Software Engineering for Technical Computing

MSc Thesis
Academic Year: 2011 - 2012

Supervisor: Dr Mark Lee Stillwell

August 2012

CRANFIELD UNIVERSITY

School of Engineering
Software Engineering for Technical Computing

MSc Thesis

Academic Year 2011 - 2012

Csaba Kerti

Virtual Machine Co-Scheduling Modules

Supervisor: Dr Mark Lee Stillwell

August 2012

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science

This thesis is submitted in accordance with the Double Degree programme regulations. Home institution: Eötvös Loránd University, Hungary

ABSTRACT

There can be a performance problem for High Performance Computing (HPC) applications when they are used with Virtual Machines (VM), because these VMs are running in a time shared environment. The main problem is that some of the processes, which run on a VM, may spend all the time that they are awake, waiting for communication from other processes that are currently sleeping or idle.

This is a commonly known problem, and it can be solved by using gang scheduling, which ensures that all the processes which are in the same group run in the same time. The gang scheduling has a practical problem, namely that it has a huge overhead by the synchronised distributed context switching. Co-scheduling is a relaxation of gang scheduling in which the distributed context switch is dispensed with and the local schedulers use available information to attempt to keep communicating processes on different machines running at the same time.

The aim of the thesis is to implement a co-scheduling CPU scheduler as a plug-in to the platform of Xen hypervisor. The implemented plug-in should ensure better performance than the original ones, when it is used in a HPC environment. This performance should be measurable by testing it with parallel HPC applications under various scenarios.

Keywords:

Virtual Machine, Xen, Hypervisor, SEDF, Co-Scheduller, Gang Scheduling Paravirtualization, Virtualization, Distributed Computing, High Performance Computing

ACKNOWLEDGEMENTS

The author would like to thank the supervisor of this project to Dr Mark Lee Stillwell. His guidance, assistance and ideas shared with the author made this thesis realisable.

Very special thanks and appreciation would like to given to the family of the author. His mother gave him so much love and energy, which helped to the author to keep going with his thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

BVT	Borrowed Virtual Time
ABI	Application Binary Interface
CPU	Central Processing Unit
CS	Co-Scheduling
HPC	High Performance Computing
HVM	Hardware Virtual Machine
MPI	Message Passing Interface
NAS	NASA Advanced Supercomputing
NPB	NAS Parallel Benchmarks
NIC	Network Interface Controller
NWC	Non Work-Conserving
OS	Operating System
PCPU	Physical CPU
PS	Proportional Share
RR	Round Robin
SEDF	Simple Earliest Deadline First
VCPU	Virtual CPU
VIF	Virtual Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor (also Hypervisor)
WC	Work-Conserving

1 INTRODUCTION

In this introduction, a brief description, of why there is a need for the Co-Scheduling Virtual Machines in HPC (High Performance Computing) environments will be explored.

1.1 Description of the participants

1.1.1 High Performance Computing

The first High Performance Computers were created in the 1960s. They were very important for the scientist, because they used them to solve large amount of calculations with computational cost much greater than an ordinary computer could handle. These calculation intensive tasks included; physical simulations, weather forecasting and nuclear weapon exploding.

The HPC contains an enormous number of processors, which able to decompose problems and solve them in a massively parallel way.

1.1.2 Virtual Machine

A VM (Virtual Machine) is an implementation of the hardware. Actually it is a layer between the hardware and the operating system. The software which is running on the VM will think that it is executed on a real computer. Obviously, if a program is running on a VM, it has worse efficiency than the same program is running on a bare hardware. However, using Virtual Machines has many advantages, especially when they are used in HPC environment.

Using virtual machines offers consolidation and increased hardware utilization. The servers (such as file servers, printing servers, etc.) are often underutilized, thus it is possible to move the applications into Virtual Machines which are using less number of hardware. This was confirmed by Govindan et al. [15] who created a virtualized environment to examine. The applications of three underutilized servers had been merged into one server using three virtual machines. The Figure 1 shows the result.

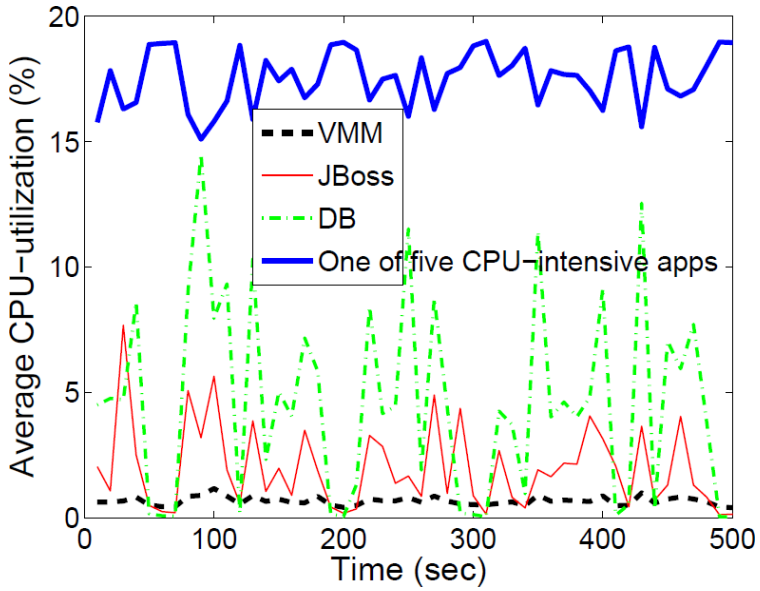


Figure 1 - CPU usage of a consolidated server [16, p. 2]

The CPU utilization of the consolidated server is still lower than 100%. The Virtual Machine Monitor (VMM) which is responsible for the administration of Virtual Machines causes a negligible overhead. A consolidated server results less amount of physical hardware, so the cost of the hardware, the maintenance and the power usage are decreasing.

Virtual Machines have further advantages like mobility, standardization, isolation and comfortable testing.

1.1.3 Scheduler

In many area of the computer science, the available resources are less than those that are required. Therefore, there is the need for a scheduler to distribute the resources among the processes, using a scheduling algorithm. This algorithm should ensure that the resources are assigned with an equal processor time for each process.

The problem is similar in a virtualized environment. The scheduler of the Hypervisor has to distribute the physical CPUs among the VMs.

1.2 Definition of the problem

Performance issues can occur when HPC applications are used with Virtual Machines. This is due to the VMs running in a time shared environment. The main problem is that some of the processes, running on one VM, may spend all of the time while it is awaken, waiting for other currently sleeping or idle processes. One example for the possible slowness within a virtualized environment is shown Figure 2.

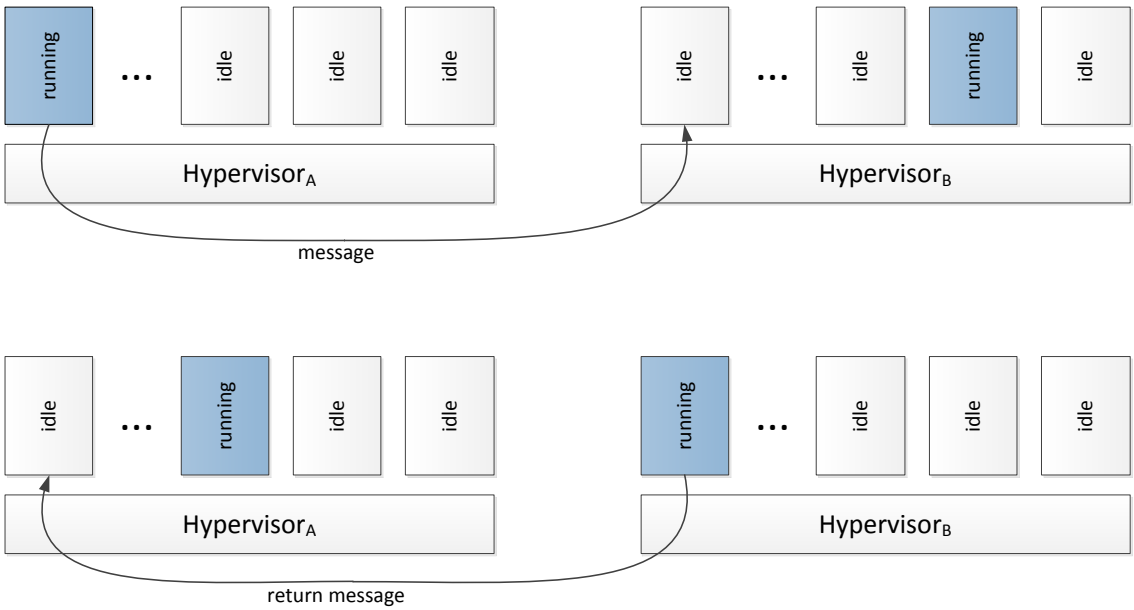


Figure 2 - Source of slowness in case of communication intensive application

Within the example in Figure 2, a process running on one VM on the Hypervisor$_A$ sends a message to another VM on the Hypervisor$_B$. The receiver VM is in idle state, therefore the message should wait somewhere at the Hypervisor$_B$ till the state of the receiver VM won't become running. The returning message similar, the answer may need to wait at Hypervisor$_A$ till the VM become running again.

Therefore, there is a need for a scheduling algorithm, so that there is not a wait between a message being sent by a VM on Hypervisor$_A$ and the message being received by Hypervisor$_B$, Different scheduling approaches are detailed in the next sub-section.

1.3 Possible Approach for scheduling

1.3.1 Gang scheduling

The previously defined problem is well known, and can be solved by using gang scheduling. Gang scheduling ensures that all the processes in the same group, run at the same time.

One practical problem for using gang scheduling is that there is a huge overhead, caused by the increased communication time of the synchronised distributed context switching. To use gang scheduling another process or a marked one needed from the group, which can transmit scheduler signals from time to time to the group.

1.3.2 Co-Scheduling

Co-Scheduling is a mitigation of gang scheduling where the distributed context switches are not used. Therefore the local schedulers need to decide about the local context switches and ensure - by using the available statistical information - that the communicating processes on different machines are running in parallel.

1.4 Chosen technology

The Xen Hypervisor [30] has been chosen for execute the Co-Scheduling experiment.

The main reasons for choosing Xen hypervisor are:

- Xen hypervisor is the most widely used for virtualization, especially in Cloud and HPC environment.
- Fastest product on the market. This is partly the merit of using the paravirtualisation technology. The Chapter 3 contains more information about the paravirtualisation technique.
- Open source software and written in C. This allows the modification of the product.

- Mature, tried and tested. It is proven to work well, on large systems like the Amazon Web Services [1].

Nevertheless, the documentation of the product is outdated and has many gaps, but the developers on the Xen developer community happy to answer any questions.

1.5 Aims and objectives

The main objective of the project is to implement an appropriate Co-Scheduling algorithm on-to a chosen platform to increase the performance. In order to reach these objectives, a few sub objectives are defined:

- To investigate and research current Co-Scheduling algorithms.
- To become acquainted with the architecture of the Xen Hypervisor.
- To implement plug-ins with the chosen algorithms in C programming language.
- To set-up cluster environment with the chosen Xen hypervisor for the performance test.
- To set up and execute appropriate benchmark applications. It is required that, at least one of them should have a large number synchronized inter-process communication.

1.6 Overview

We will begin this thesis with a literature review into the different Scheduling algorithms; Gang Scheduling, Co-Scheduling and Xen Hypervisor. We will then go onto the present the background on the Xen Hypervisor, its architecture, schedulers and how the communication works (Chapter 3). Following this a description of the implementation of the chosen algorithm will be detailed (Chapter 4). In chapter 5, a brief description of the setup of the environment will be detailed. Also within chapter 5 the results of the implemented algorithm and evaluation the benchmark test will be presented. Finally, in chapter 6, the conclusion of the study and future work will be detailed.

2 LITERATURE REVIEW

Within this chapter we will first detail the different scheduling methods before going on to explore benchmarking.

The literature uses the co-scheduling word in many different aspects (co-scheduling, Co-Scheduling and coscheduling). The "co-scheduling" form will be used in this paper, but when another paper is referred, the form used there will be used here too. The situation is similar with the word "preemptive". The literature uses the pre-emptive form as well.

2.1 Scheduling

Tanenbaum [2, p. 145-163] presents the fundamental schedulers and describes the general requirements for the schedulers in his book. The scheduler of the OS ensures smooth operation between processes. Usually there are more processes than the number of processors, so not every process can be in running state. The main function of a scheduler is to alternate the processes, to achieve the illusion that the processes are working in parallel and every process owns a CPU. The following diagram shows the states of the processes and their transitions from the perspective of Tanenbaum.

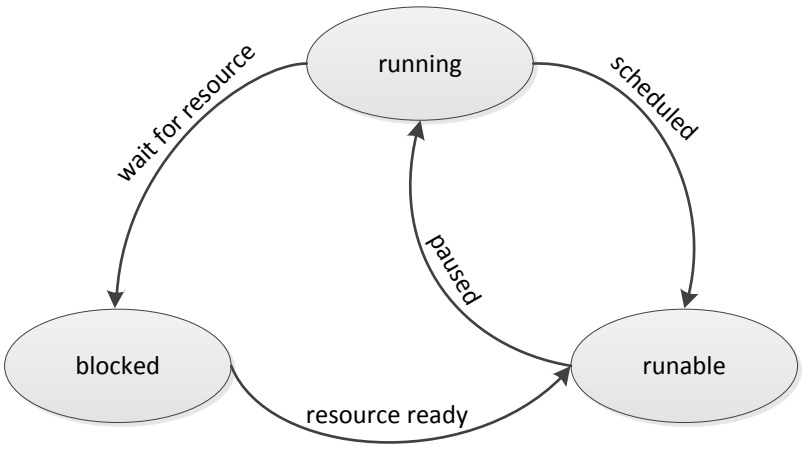


Figure 3 - states of the processes and their transitions [2, p. 90]

The main purposes of the schedulers are:

- Fairness: the need of equal amount of CPU time for each process.
- Policy enforcement: the operation of the scheduler fit to the chosen algorithm.
- Balance-keeping: the scheduler should ensure maximum resource utilization.

Tanenbaum [2] divided the schedulers into three classes, depending on what type of systems are using them. Several scheduler algorithms have been described for each scheduler classes:

- **Batch** – scheduling algorithms include: First-Come First-Served, Shortest Job First, Shortest Reaming Time Next.
- **Interactive**: Round-Robin, Priority Scheduling, Multiple Queues, Shortest Process Next, Guaranteed Scheduling, Lottery Scheduling.
- **Real time schedulers**.

2.1.1 Gang scheduling

2.1.1.1 Ousterhout matrix

Ousterhout [21] created the basics of the Gang Scheduling and introduced the idea of Co-Scheduling expression. The author observed that the parallel processes of applications from the 80s are not independent. If the processes of an application are communicating with each other, then it is matter that when they are doing it. Ousterhout [21] study examined how the multiprocessor environment behaves when there are intensive inter-process communications.

The led to the idea of “pause time”, where if a process become blocked due to waiting for an event to occur, then the waiting time should predetermined. Once the event has occurred, then the process can continue its work.

Ousterhout raises the problem of trashing. In a system, half of the processes are running only in odd time slices and the other half of them in even time slices

If the two task forces are interacting with each other, it is possible that the task force which is sending messages to the other task force will become blocked while it waits for the response from the other, descheduled task force. This gives an upper limit. The processes can communicate as many times as the system executes contact switches.

Ousterhout offers a solution for this problem by arranging the processes into groups. The processes which are communicating with each other should belong to one group, and all process should belong to only one group. A group is Co-Scheduled, if all of its processes are running at the same time. To schedule the groups, the author uses a special matrix which is organized in the following way. The processors of the system are associated to the columns of the matrix. The groups can be allocated into a row, if there is enough space for all processes in the group as the Figure 4 shows. The scheduling uses the simple Round Robin mechanism. First, it executes the first row. In the next time slot the following row will be scheduled and this continues cyclically.

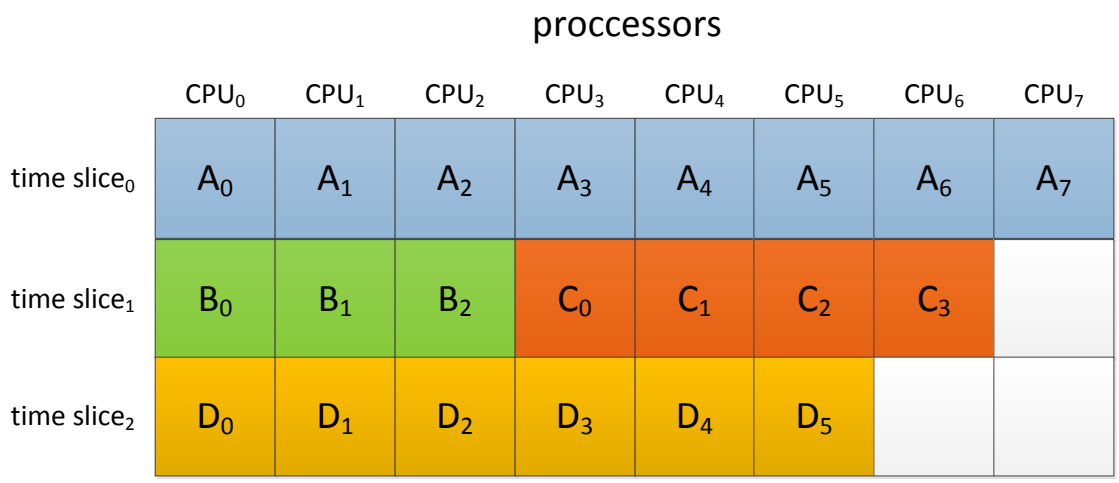


Figure 4 - The Ousterhout matrix

Alternate selection is used from the actual column, when there are empty or idle cells in the scheduled row. This is necessary for the sake of avoiding inactive processor times. The Continuous and the Undivided algorithm has been introduced to ensure better utilization and to decrease the number of rows in the Ousterhout matrix.

2.1.1.2 Flexible Co-Scheduling

The aim of Frachtenberg at el. [13] was to create a gang scheduler which mitigates the load imbalance of applications. The main causes of load imbalance are the heterogeneity of hardware and the imbalance of the application.

The algorithm – Flexible CoScheduling (FCS) – classifies the processes into three groups:

- CoScheduling(CS): These processes communicate a lot and the efficiency is increasing if they are gang-scheduled.
- Frustrated(F): These processes communicate a lot but due the load imbalance they cannot utilize their CPU time.
- Don't care(DC): The processes which are rarely synchronize.

The following figure shows how the classified classes are located relatively to each other.

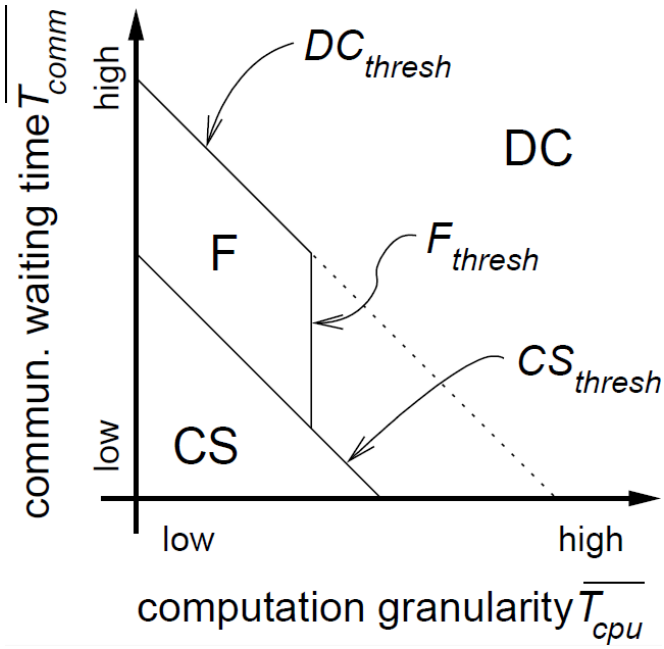


Figure 5 - Classification of processes [13, p. 5]

The scheduler based on the traditional gang scheduling, namely a central manager sends multi context-context-switch messages to the other nodes in the

system. The nodes decide to execute the context-switch, based on some local information. If the process of the message is a CS process, then the context-switch should be executed and the actual process became interrupted. In other cases the context-switch will be refused and the process will be executed later. The scheduler use spin-block for the F processes to mitigate the number of context-switches. The DC processes are useful to fill the gaps between other processes.

The solution had been compared with other type of schedulers and it found to be competitive.

2.1.1.3 Adaptive time / space sharing

Sodan et al. [27] had demonstrated a gang scheduler which uses an adaptive resource allocator. The algorithm uses the Ousterhout matrix [21] as a basis to allocate the processes. The allocator distinguishes between the size and types (moldable, malleable and rigid) of the processes. The algorithm shrinks or expands the tasks depending on the actual workload of the system. This decision can be made by using the speed curve or by using node numbers (N_{min}, N_{opt} and N_{max}) and the corresponding efficiency values (E_{min}, E_{opt} and E_{max}). They used two backfilling techniques and the expanding of malleable jobs to fill out the gap in the allocation matrix.

Many variations have been implemented, and the scheduler which uses priorities, system-load adaptation and EASY backfilling techniques offered the best performance. A further way to improve the efficiency has been to use a fragmentation adaptation. This algorithm has the best performance when the maximum multiprogramming level (number of time slices) is one. The result showed that this adaptive SCOJO algorithm performs better than the normal gang scheduler.

2.1.1.4 Buffered Co-Scheduling

A further variation on the Co-Scheduling, was developed by Fabrizio Petrini and Wu-Chun Feng [22]. They used both gang-scheduling and implicit Co-Scheduling techniques. The two main features of their work are the

communication buffering and strobing. The communication buffering technique accumulates the network messages of the process and delivered them at the end of a given interval. This reduces the network overhead, although non-blocking communication is needed to gain efficiency. The communication buffering is combined with the strobing technique, which is basically a gang scheduling. There is a special master node, which sends a “heartbeat” to the workers, to sign the end of the intervals. According to the authors, this solution uses only low level features of the network cards for the generation of the heartbeat, which ensures the low overhead. The executed measurement showed that the solution could gain better resource utilization, especially when the communications between the processors are intensive.

2.1.2 Co-Scheduling

2.1.2.1 Co-Scheduling Strategies for Networks of Workstations

The purpose of Shailabh Nagar et al. [25] was to create a comprehensive comparison of implicit Co-Scheduling algorithms. They created a number of new mechanisms; Periodic Boost (PB), Periodic Boost with Spin Block (PB-SB), Spin Yield (SY), Periodic Boost with Spin Yield (PB-SY) and Dynamic Co-Scheduling with Spin Yield (DCS-SB). They gathered them together with the already existing Co-Scheduling algorithms and implemented them to be able to execute the measurements. An overview of the different algorithms implemented is shown in Table 1.

	Busy wait	Spin Block	Spin Yield
No Explicit Reschedule	Local	SB	SY
Interrupt & Reschedule	DCS	DCS-SB	DCS-SY
Periodically Reschedule	PB, PBT	PB-SB	PB-SY

Table 1 – Organized Co-Scheduling algorithms [25, p. 3]

The column title details what the scheduler should do, when a process started to wait for a message, while the row title details what to do, when a process receives a message. The algorithms inside the table are the following:

- **Local**: The waiting process spins, until the message doesn't arrive. Probably it is the worst algorithm. It is used as a baseline of the measurement.

- **Spin-Block (SB)**: The waiting process spins for a certain amount of time and if the answer arrives before this time elapses, then it saves the context-switch. The fixed time should be lesser than the double of the context-switch time, in order to achieve better scheduler performance. The likelihood of that two communicating process are running in the same time for a while, increases.

- **Dynamic Coscheduling (DCS)**: When a message arrives, the priority of the receiving process is increasing and the actual process becomes pre-empted. The disadvantage of the algorithm is the intensively communicating processes. In this case, the algorithm generates context-switches frequently.

- **Dynamic Coscheduling with Spin Block (DCS-SB)**: The basic DCS can waste lot of time with waiting for a message, spinning. The DCS-SB combines the advantages of the two algorithms, and cures the problem of the basic DCS.

- **Periodic Boost (PB) and Periodic Boost with Timestamp (PBT)**: If the processes are intensively communicating, then the DCS algorithm generates numerous context-switches. The Periodic Boost cures this problem. The scheduler periodically checks the processes and increases the priority of the worthy processes. The strength of the algorithm is that a larger number of more complex mechanisms can be used to locate the most worthy processes. PB increases the priority of the first process which have pending messages, using the round-robin method to locate it. The Periodic Boost with Timestamp chooses the process with the most freshly arrived waiting message. This also increases the possibility that

two communicating processes will be running at the same time for a while. Both versions have been implemented easier, then the DCS or the PB.

- **Periodic Boost with Spin Block (PB-SB)**: Simple combination of PB (or PBT) and SB. It has inherited the overhead of PB and SB as well.
- **Spin Yield (SY)**: This is an alternative of the Spin Block algorithm. The waiting process spins for a certain amount of time and if the answer doesn't arrive before this time elapses, then the priority of the process will decrease and another pending process will be boosted.
- **Dynamic Coscheduling with Spin Yield (DCS-SY):** Combination of Dynamic Coscheduling and Spin Yield.
- **Periodic Boost with Spin Yield (PB-SY):** Combination of Periodic Boost and Spin Yield.

The authors implemented these 9 algorithms and used the NAS Parallel Benchmark [20] to execute the measurements. According to the results of the measurements, if the processes of the experiments have low communication intensity, then there is no significant advantage of the algorithms. Nevertheless as the communication intensity is increasing, the implemented schedulers became more efficient. In overall, the Periodic Boost is looking to be the most usable algorithm. It offers good response time and throughput. Furthermore, all of the three algorithms which use the Dynamic Coscheduling algorithm are performing reasonably well.

Anglano [3] tried to help to improve the performance of the frequently communicating application, using implicit Co-Scheduling techniques to avoid gang scheduling. The study extends the previous work of Nagar et al. [25] who implemented and tested the performance of nine implicit co-scheduling algorithms. The authors created three more algorithms and grouped them with the earlier strategies, to execute a comprehensive benchmark.

The common part of the three new algorithms is the Block behaviour. When a process starts waiting for a message, it blocks immediately. They inserted this algorithm into the 3x3 matrix of the previous study, which combines the behaviour of the message waiting and the action handling algorithms. The Table 2 shows the extended matrix.

	Spin	Block	Spin-Block	Spin-Yield
None	Local	IB	SB	SY
Immediate Boost	DCS	DCS-IB	DCS-SB	DCS-SY
Periodic Boost	PB	PB-IB	PB-SB	PB-SY

Table 2 - Implicit Coscheduling Strategies [3, p. 3]

Four different workloads had been created to the execute test cases. The final conclusion of the study that co-scheduling strategies have great influence to the performance, but there was no single co-scheduling algorithm which has general solution for all the workloads.

2.1.2.2 Implicit Co-Scheduling Strategies for Networks of Workstations

The article of Angela C. Sodan and Lei Lan [26] presents an algorithm which can effectively co-schedule processes on Hyper-Threading (or standard) processors. The study was conducted after the Hyper-Threading processors of Intel had released. This technology gives possibility for two processes to run parallel on a processor without context switches. The algorithm takes advantage of the possibility of Hyper-Threading, but it can work well with many-cores processors. However, within a many-core processor, these processes have significant competition for the common resources, such as I/O, network, memory and CPU resources.

The Sodan and Lei Lan [26] created three process classes (CPU, network and disk) to organise the processes. Each process should belong to one of the three classes. It is assumed that the processes are monitored time to time, so this decision of classification can be achieved dynamically. An observation has been

made, which examine how the process classes interfere with each other. According to the results, if two processes - which run at the same time - are from different process classes, then the interference is minimal. However, if the processes are from the same classes, then the interference is significantly higher (except CPU vs. CPU).

The solution tries to find the best matching process to the current process from the process queue. If there is no free execution unit, then the algorithm tries to find a matching process from the running processes. Using this procedure the algorithm achieves significantly better response time than the default scheduler.

2.2 Xen

2.2.1 Background of Xen

Both the book of William von Hagen [32] and the book of Chris Takemura & Luke S. Crawford [7] are essential for system administrators. These studies are beginning with a detailed overview of the technology and the architecture of Xen. The books are also presenting how to install and configure varying guest domains, setup networking, manage storages, etc.

The book of David Chisnall [9] gives a very deep view of the architecture of Xen. This book has written especially for developers. It covers how the schedulers are working, how the messaging is working on the event channels, handling the memory etc. It gives examples and source code for every chapters of the book.

2.2.1.1 Xen and the Art of Virtualization

The creators of the Xen hypervisor [4] had created this article to present their work. The Xen hypervisor detailed within this article was the pre-release state, close to the version 1.0. This version only contains the Borrowed Virtual Time (BVT) scheduler which is unavailable in the latest versions of Xen.

The aim of the authors was to create a remarkably efficient virtual machine monitor, which is able to run 100 virtual machines. They had four main goals:

- The applications of guest operations systems should not be modify to be runnable.
- Support the modern, multi-application operating systems
- The usage of the paravirtualization implementation against the full virtualization (because to support the high performance)
- The effects of the resource virtualization from the domain operating systems should not be hidden (to reach better performance and improve correctness).

To achieve the paravirtualization, three main part of the guest operations system should be modified:

- CPU (Exceptions, Protection, System calls, Interrupts, Time)
- Memory management (segmentation, paging)
- Device I/O (Network, Disk)

Two main communication forms exist between the Xen VMM and a domain. The first one is the domain to Xen direction, which uses (synchronous) the hypercalls. The second communication form is the Xen to domain direction, which uses asynchronous events mechanism. A special I/O ring had been created for this communication, which helps to improve its efficiency.

Furthermore the authors describe the details of the time handling, the virtual address translating, the physical memory partitioning, the network and the disk handling.

The authors had done many performance tests on the established hypervisor. The results showed that the Xen hypervisor is significantly better than its rival hypervisors, like VMware and User-Mode Linux. Furthermore the performance of the modified Linux, - which is run on the Xen hypervisor - is very close to the native Linux systems. For these reason, we have decided to use this method within this study.

2.2.1.2 Comparison of the Three CPU Schedulers

Chereskova et al. [6] performed a study to compare the performance of three schedulers of the Xen v3.0 hypervisor.

First, some technical terms had been clarified, like: proportional share, fair-share schedulers, work-conserving, non-work conserving, preemptive and non-preemptive. The features and behaviours of the different implemented schedulers (BVT, SEDF and Credit scheduler) had been characterized.

Three performance applications (web server, network throughput and disk I/O) were created by the authors to measure the performance of the schedulers. They used various parameters (context switch allowance or period and Dom0 weight) for the performance testes.

In most cases part of testes the BVT had the highest throughput. The SEDF scheduler worked more efficiently with non-work conserving than with work conserving. In total, usually the Credit reached the smallest throughput from the three schedulers.

The CPU allocation error had been evaluated in case of SEDF and Credit scheduler. The SEDF scheduler usually under-allocates, while the Credit scheduler over-allocates. However, the allocation error of the SEDF scheduler was much smaller than that of the Credit scheduler.

Another advantage of the Credit scheduler is that it supports global load balancing, which can reach better CPU allocation in case of multiple CPUs. The measurements have been executed in a multiple CPU environment as well, where usually the Credit scheduler reached the smallest throughput.

2.2.2 Creating new Scheduler to the Xen hypervisor

2.2.2.1 Communication-aware CPU Scheduler

Govindan et al. [15, 16] identified two shortcomings of the actual Xen hypervisor. The schedulers do not count the increased number of communication active network applications and the measure of the overheads of I/O virtualisation can sometimes be inappropriate. A communication aware

CPU scheduler and CPU usage bookkeeping mechanism had been implemented for the Xen hypervisor to solve this problem. This communication-aware scheduler prefers those VMs which have intensive communication slightly against other. The study gives a description of the structure of the Xen VMM and the scheduler of the Xen hypervisor. During the communication of the Virtual Machines, three types of scenarios had been identified when the schedulers of Xen induce delays:

- There are two delays at Domain0: The time between the physical NIC get the message and when Domain0 is scheduled. Another delay is the duration between when the sender VM copies the message into the transmission ring and when the Domain0 is scheduled.
- Delay associated to receiver: The time between the Domain0 set up the event channel and when the receiver VM obtaining the schedule.
- Delay associated to sender: The time between the sender VM sending the message packet and when a domain gets the package.

The first two delays can be prevented by using a scheduler which prefers the domains which have intensive communication. If there are more from these domains in a moment, the scheduler should choose the domain which receives the most messages. The third event not foreseeable, but the authors tried to use a prediction using the previous behaviour of the domain. A bookkeeping technique had been used to register the incoming / outgoing packages to help the scheduler to choose the mostly communicating domain.

The experimental test shows the usage of this scheduler has a significant performance improvement and reduces the context switches in an intensively communicating server environment (typically web-server environment).

2.2.2.2 Real–time hypervisor scheduling in Xen

The purpose of Sisu Xi at el. [36] was to create such VM schedulers to the Xen hypervisor, which supports real time applications. In contrary to the RT-Xen schedulers, the built-in schedulers of Xen are aimed at increasing the utilization.

The quantum of the VCPUs had been changed from 30 milliseconds to 1 millisecond to decrease the response time. This number is based on an empirical study, where several measurements were executed with different quantum between 1 and 10 milliseconds. Four different scheduler algorithms had been implemented: Deferrable Server, Periodic Server, Polling Server and Sporadic Server. The authors extended the two main schedulers of Xen with two new schedulers – as visible on Figure 6 -, which were created to cover these algorithms. The schedulers are based on three parameters: priority, budget and period. The budget determines the maximum running time of a VCPU in each period. Three of the algorithms uses fixed periods, while the Sporadic Server not. The algorithms main difference is how they handle the queues of the schedulers which store the CPUs. Three queues are used by the schedulers:

- RunQ: Stores those VCPUs by priority, which state is runnable. Every time, when the main scheduler functions runs, it picks the first VCPU to schedule it.
- RdyQ: The VCPUs with enough budgets to run are here, but they have nothing to do.
- RepQ: Stores those VCPUs which don't have enough budget to run and they need to be replenished.

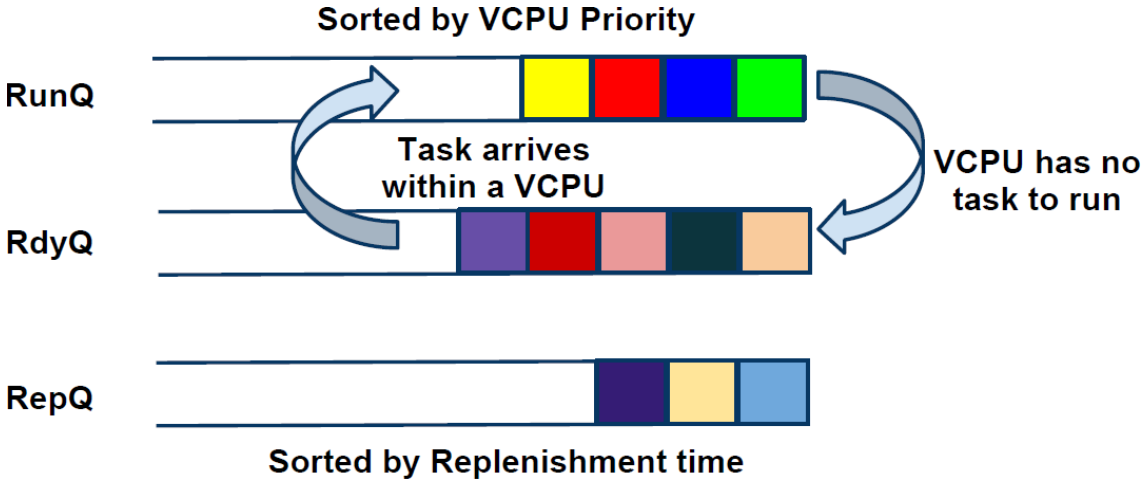


Figure 6 - Queues of the RT-Xen in work [34, p. 5]

The evolution of the four new schedulers have been executed and compared with the two main schedulers of Xen hypervisor. The new schedulers makes more than two times more overhead then the built-in SEDF and more than four times than the Credit scheduler of Xen. However the schedulers of RT-Xen execute the main scheduler function two times more efficiently than the SEDF and about six times more than the Credit one. The real-time schedulers are providing better response time under these conditions. All things considered, the Deferrable Server looks to be most applicable, while the efficiency of the Periodic Server is the worst when the system is overloaded.

The authors allow us to download the source code modifications and follow the changes of their project [24].

2.2.3 Performance and monitoring of Xen

2.2.3.1 Performance Evaluation of the CPU Scheduler in XEN

Xianghua Xu et al. [35] made a performance evaluation of the Credit scheduler of Xen. The study analysed the parallel running of different types of guest domains, focusing on how they impact to each other performance,

The authors created five different types of test applications:

- CPU consuming calculation
- Network intensive application
- Disk I/O intensive program
- Typical web server program
- Database transactions simulating application

The studies showed how these applications impact to each other's performance, under various domain weight parameters. The study provided the following results:

- A network intensive application becomes very sensible if it is running in parallel with a CPU intensive application. As the weight of the compute

intensive domain increasing, the throughput of the bandwidth intensive application is decreasing.

- When a CPU and a disk intensive domains are running in parallel, the varying weight of the disk I/O intensive domain has nearly no impact to the CPU intensive domain.
- If a web server or a database application runs in parallel with a disk I/O or a calculation intensive domain, then it is apparent that the reply time of the web server and the processing time of the database application are increasing as the other domain gets more weights.
- The study helped to understand the impact of parallel running Xen VM deeply, and informs system administrators of the best way to setup the domain parameters, when they use the Credit scheduler in the Xen hypervisor.

2.2.3.2 Performance Interference

The purpose of Xing Pu et al. [23] was to investigate the performance of the interfering Virtual Machines on the Xen hypervisor. Analysis was executed to discover the effects of the parallel execution of the network or CPU bounded VMs to each other and to the Domain0.

A detailed description was presented relating to how the messages travelled from the Network Interface Controller (NIC) through the Xen VMM and the Driver Domain to a Guest Domain and back.

The first step of the measurement was to create a base case with a single guest domain. It was important for the future comparison and for making conclusions. Several workloads had been created for the measurement and were split into two groups (CPU and Network bounded workloads).

During the measurement, two guest domains were used with the previously created workloads. A combined score, based on the throughput was used to evaluate the performance of the guest domains. The test has four important results:

- Running two network-bound VMs on the same hardware can lead to high overhead, due to the increased context switching and the high number of cache misses.
- Executing two CPU-bound VMs results high CPU competition for the guest and for the driver domains as well, but the interference is less than in the two network-bound VMs case.
- The combination of a CPU-bound and network-bound VMs results in the lowest competition for the resources and this combination deliver's the highest performance, visible on Figure 7. The (N_1, N_2) annotation means that the VM_1 has N_1 while the VM_2 has N_2 network traffic. (1 = low network traffic; 100 = high network traffic).

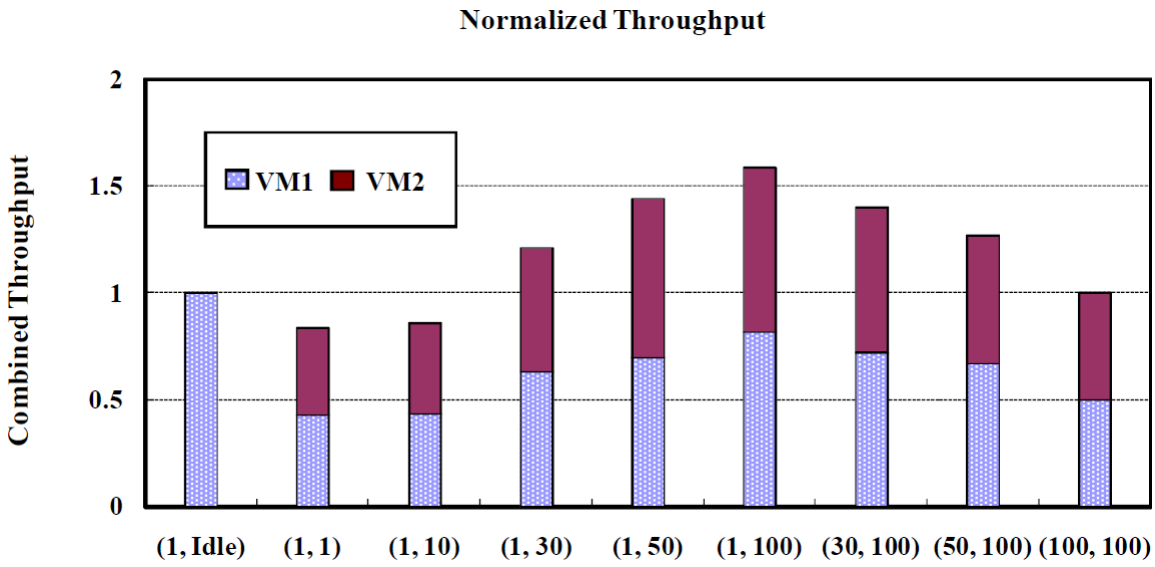


Figure 7 - Combination of a CPU and a Network Intensive VM [23, p. 3]

- The interference depends from the driver domain as well, because it is responsible for the switches and for the I/O processing.

2.2.3.3 Bottleneck detection

The purpose of Min Lee et al. [18] was to create a monitoring and tuning tool to determine the bottlenecks of Media Applications which are runs on Xen hypervisors. The performance challenges of the scheduler of the Xen VMM had been analysed when real-time applications run on it. The tool determined the

performance bottlenecks which was the network I/O and the scheduler. It is possible to modify the parameters of the schedulers to gain better performance, but it is usually hard to accurately measure and based on guessing.

The paper gives a general review from the Xen hypervisor and the Credit scheduler. They mention that the Credit scheduler differentiates three type of priority: over, under, boost and describe how the scheduler handles them in a three different run-queue.

The authors created a monitoring tool called XenTune for monitoring and tuning the scheduler using the running results. The monitor uses the XenTrace tool of the Xen hypervisor, which produce event logs from time to time. The major parts of the XenTune are an event processor, a metrics analyser and the Scheduler Parameter Tuning. The architecture of the XenTune is visible on the following diagram.

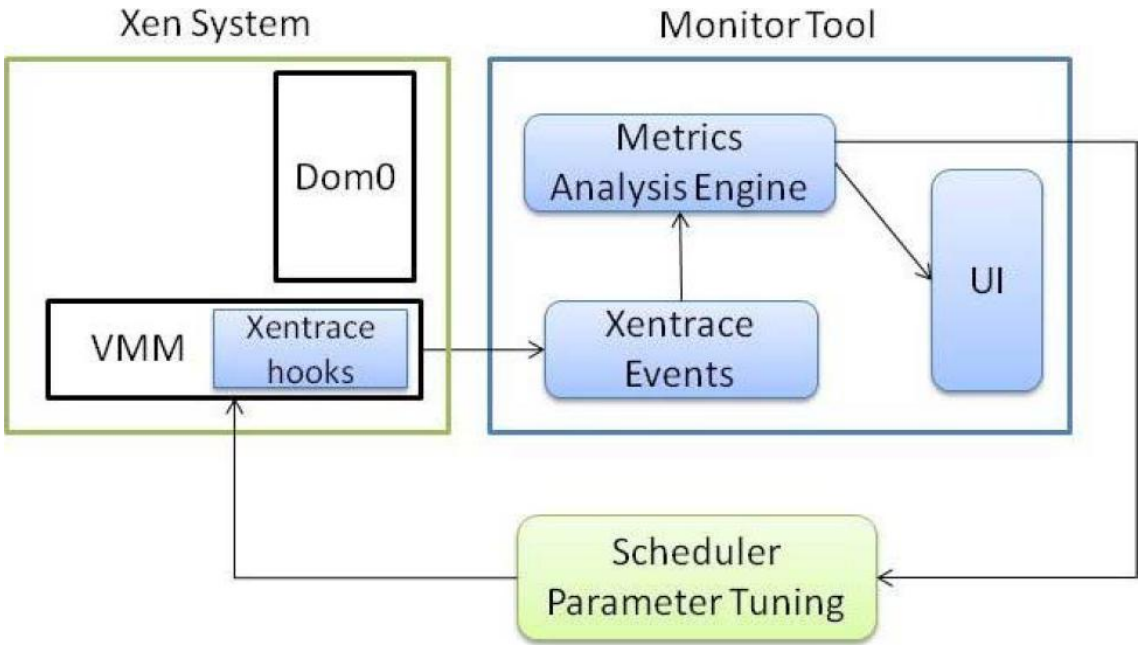


Figure 8 - XenTune monitoring architecture [18, p. 3]

The monitor collects the following metrics to explore the bottleneck of the scheduler: number of execution, average waiting time in the appropriate queue, average running time on a CPU and the total scheduled time.

The behaviour of different schedulers had been measured, like default Credit scheduler, Credit scheduler with weights, Credit scheduler with pinned CPU and with the authors' modified scheduler. The result of the experiment showed that default Credit scheduler worked insufficiently even if they used weights. Pinning performed better, but in this case, other VM-s got less CPU time, and the pinned CPU wasn't fully utilized. The media application on the modified scheduler performed similarly than the pinned one, but this scheduler shared the CPU.

2.2.3.4 Measuring CPU Overhead for I/O Processing

Ludmila Cherkasova and Rob Gardner [5] made a monitoring system for the Xen hypervisor to measure the overhead and usage of the server CPU under the network I/O. Network packages were created with size between 1kb and 70kb for the experiment. The authors measured the throughput (req/sec), the network I/O (Kb/sec), the CPU utilization of the Domain0 and the amount of memory exchanges (pages/sec).

2.3 Benchmarking

2.3.1 The NAS Parallel Benchmarks

The study of David H. Bailey, et. al [11, 12] gives a general overview of the NAS (NASA Advanced Supercomputing) Parallel Benchmarks (NPB) [20]. It was created by NASA, fur use as a measurement tool for parallel HPC projects. The NPB originally contains 8 benchmark programs (later it has been extended with 3 more) which solve their numerical computation problems in a distributed way.

- Kernel problems
  - **Embarrassingly parallel (EP)**: This benchmark differs from the others. There is no interprocess communication at all. Only at the beginning, when the master node distributes the problem to the worker nodes and at the end, when the master collects the results.
  - **Multi Grid (MG)**: The Multi Grid method requires huge interprocess communication.

    - **Conjugate Gradient (CG)**: Calculates the estimated eigenvalue of a positive definite matrix. It requires high interprocess communication.
    - **Fourier Transformation (FT)**: The solution solves a 3D partial difference equation. It requires high interprocess communication.
    - **Integer Sort (IS)**: Implements a distributed sort of an array with integer numbers.
- Three Computational Fluid Dynamics (CFD) applications, which are the most computationally intensive parts of the CFD problems
    - **Block Tridiagonal (BT)**
    - **Scalar Pentadiagonal (SP)**
    - **Lower-Upper (LU)** with symmetric Gauss-Seidel

The programs are created in a way, which makes the size of the problem to be solved settable. The following problem size classes were created in 1991:

- Class W: Implementation for a single workstation.
- Class A: For middle sized parallel computer, as defined in 1991.
- Class B: For large parallel computers, as defined in 1991. The computations requirement is about 4 times bigger than the problem size of Class A.
- Class C: It is defined in NPB 2.2. The size of the problem about 4 times bigger than the problem size of the Class B.
- Later further classes will be introduced (Class D, E and F). The problem size of a class is 16 times roughly bigger than the previous one.

In the NPB 3.0 the MPI [29] implementation of each problem was released, and we will use them in this paper.

The number of interprocess communications in a solution of the problem is very important, because as Shailabh Nagar et al. [25] showed in their work, the co-scheduling schedulers can perform well only in an application where the communication is intensive between the processes.

S. White et. al [31] implemented the five kernel problems for the PVM (Parallel Virtual Machine) and measured the communication of each problems. The result is the following:

- EP: absence of communication
- MG: The solution spent between 42%-77% of its time with communication (depending on the number of problem size and the number of processors)
- CG: Between 67% and 77%
- IS: Between 90% and 98%
- FT: Between 48% and 70%

In the study of Shailabh Nagar et al. [25], they measured the network communications of three NPB problem (EP, LU and MG). They experienced that EP takes less than 1% of its time with communication. LU has a lot of small messages and its communication intensity is roughly 16%. The most communication intensive problem from the three is the MG, which takes about 26% runtimes with communication.

3 BACKGROUND

Virtualization has been used for nearly four decades, with explosive growth in the last ten years. It becomes more and more popular due to the enterprise computing and the appearing Cloud computing technology. Companies have discovered the benefits of virtualization and so it is starting to be widely used. The advantage of the virtualization is mentioned in the Introduction. Among other things, it helps to increase hardware utilization and reduce cost and complexity. Despite of its benefits, the virtualization technique has some drawbacks, such as the decreased efficiency, increased overhead, increased risk with the centralization.

David E. Williams and Juan Garcia formulated the essential of the virtualization:

> *"A framework or methodology of dividing the resources of a computer hardware into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time sharing, partial or complete machine simulation, emulation, quality of service, and many others."* [10, p. 8]

Many type of virtualization exists, such as the hardware level virtualization, Virtual Machines (VM) of high level programming languages (Java, .Net) and Virtual Machine Monitors (VMM).

In our work, the virtualization means virtualized computers at the OS level. This is reached by the hypervisor (VMM) software, which is sitting on the OS or running directly on the physical machine. It allows multiple and different type of operating systems running on a single physical machine, as the **Hiba! A hivatkozási forrás nem található.** shows. It schedules the VMs and provides virtual resources to the VMS.

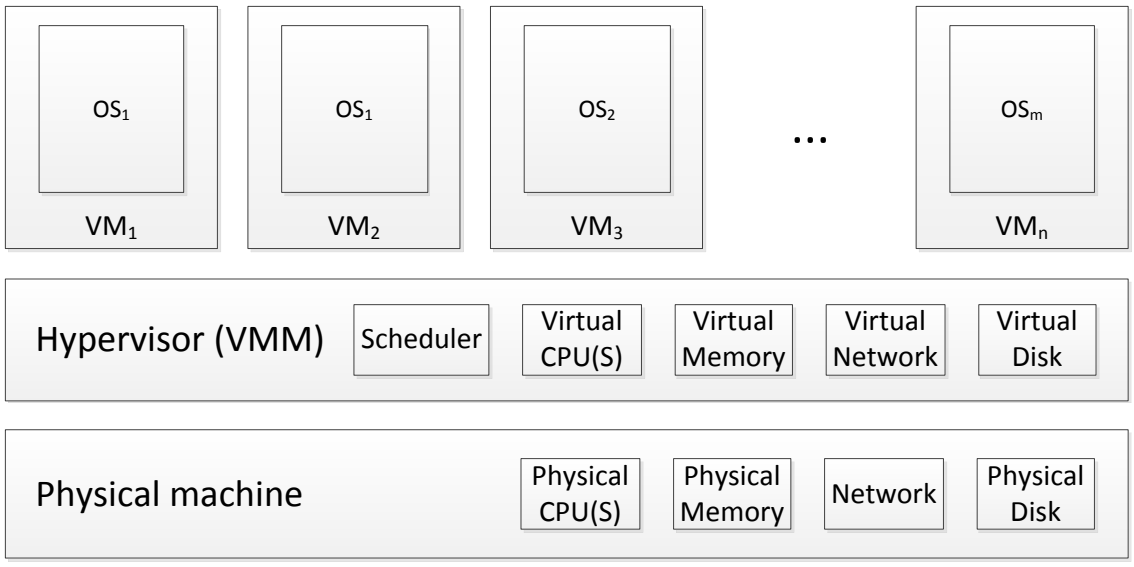


Figure 9 - General architecture of Virtualized computers

3.1 Xen

3.1.1 History of Xen

The development of the Xen hypervisor is started in the University of Cambridge by Ian Pratt and Keir Fraser in the late 90s. It was first published in 2003. The origin of the name Xen is not entirely clear. Some people derives the name Xen from the “neXt gENeration virtualization” expression. In contrary, according to Jon Crowcroft [17], one of the founder developers, the name was originated from the book of “Zen and the Art of Motorcycle Maintenance: An Inquiry into Values” by Robert M. Pirsig. The book has some philosophical thoughts connected to the “real” and to “virtual” and to “para-virtualization”.

In the beginning, the hypervisor was supported by the XenSource inc., but in 2007, ownership was changed to Citrix System. It became widely used in the HPC and Cloud computing environment. It has been always remained an open source software and such big companies are standing behind it like Intel, Oracle, Novell and Hewlett-Packard..

3.1.2 Overview

The work of the creators of the Xen hypervisor [4] and Ludmila Chereskova et al. [6] has described a full overview of Xen, which can be found in the Literature Review part of this work. However this chapter gives a briefs overview about the Xen.

Xen uses the paravirtualisation technique. This gives improved efficiency – near to the native performance -, because the running OS is laying closer to the physical hardware. It gives a software interface to the VM, which is similar to the hardware interface. The drawback of this technology is that the drivers of the OS of the VM are needed to be modified.

While the drivers of memory management, CPU and device I/O needed to be modified under paravirtualized mode, it was a design principle that the ABI (Application Binary Interface) should not need to be modified. In this way the applications which are running on the OS do not need to make changes.

The full virtualization is the opposite of the paravirtualisation, where the VMM simulates the underlying hardware. Xen can work also with Hardware-assisted full virtualization mode. The OS does not need to be modified, because the operation of the default device drivers is appropriate, thus it has the advantage of using more OS on the VM.

3.1.3 Architecture

The architecture of Xen built up with three essential elements (Hypervisor, Doamin0 and Guest domain).

3.1.3.1 Hypervisor

This is a low level VMM, which is running directly on the hardware. It is loaded under boot time, before the guest domains. The domains are able to reach the physical hardware through the Hypervisor. The hypervisor contains the scheduler which controls the execution of the domains.

Xen introduce an abstraction to the hypervisor. It separates the implementation from the administrator decisions which was left to the domain0.

3.1.3.2 Domain0

Other names: dom0, domain0 or driver domain.

The Domain0 is working as a mediator between the guest domains and the hypervisor as the following figure shows it. It implements the modified driver which allows reaching the physical hardware directly. It is responsible to manage (create, start, stop, etc.) the guest domains.

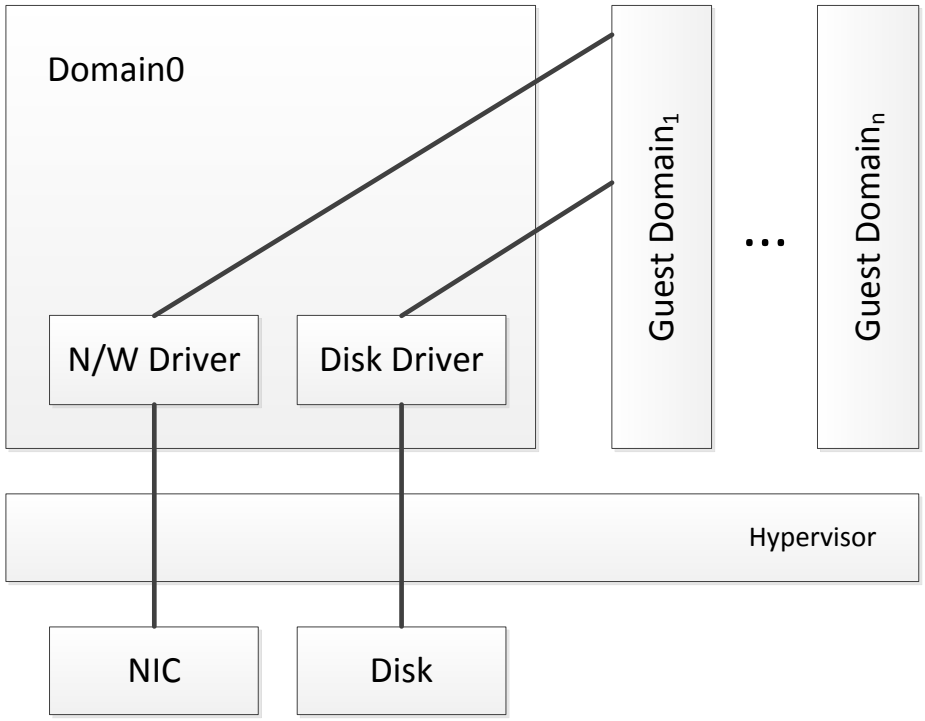


Figure 10 - Schematic architecture of Xen [6, p. 3]

3.1.3.3 Guest domain

Other names: guest, domU or domain.

The guest domains run the applications with the real content. If they are running in paravirtualized environment, the drivers of the OS are needed to be modified as well.

Each guest domain can get as many VCPUs as many physical CPUs are located in the hardware.

3.1.4 Communication

In their work, Shailabh Nagar et al. [25] have written that the co-scheduling algorithm can help only to the communication intensive processes, thus it is essentially important to understand, how the communication is working inside the Xen.

The guest domains reach the network in a virtualised way. The Domain0 is responsible for the communication of the guest domains with the outer world. Every guest domain contains a Netfront driver, which supports the virtualized communication. The Netback driver of the Domain0 is a bridge between the guest and the physical network hardware.

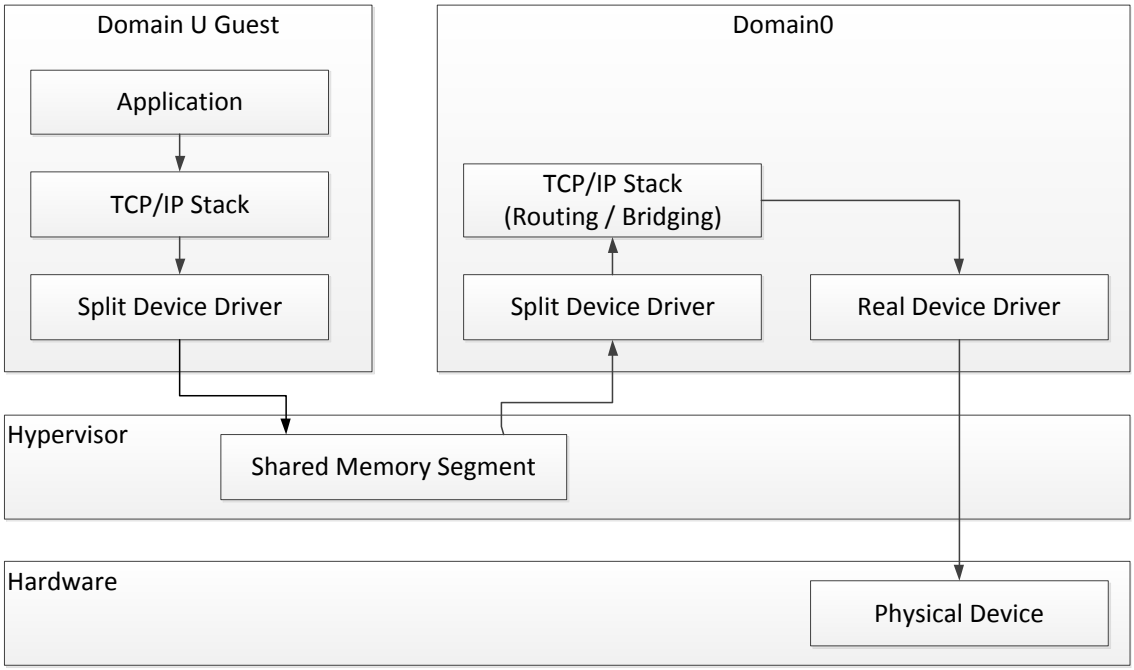


Figure 11 - The path of the network packet from the guest domain to the physical network device [9, p. 20]

Xen hypervisor uses two mechanisms to communicate between the hypervisor and the domains. The event channels are used for notification, while the I/O rings are responsible for data transferring.

3.1.4.1 Event Channels

The event channel is an efficient, asynchronous notification between the guest domain and the Domain0 and between the Domain0 and the Hypervisor. The events on the event channel are substituting the hardware interrupts.

Often some data belongs to an event, thus event channels are usually used with I/O rings.

3.1.4.2 I/O Rings

I/O rings are created for asynchronous data transferring between the hypervisor and the guest domains. An I/O ring is a circular queue. One domain puts a request in to the I/O ring, while the other removes it and puts back the answer. It can be reached by using producer and consumer pointers on both sides, as Figure 12 illustrates it. Each I/O ring contains a buffer for descriptor storing. The descriptors are containing the data directly. The data is stored on another place, due to it allows the zero-copy mechanism implementation for the I/O rings, which results fast communication.

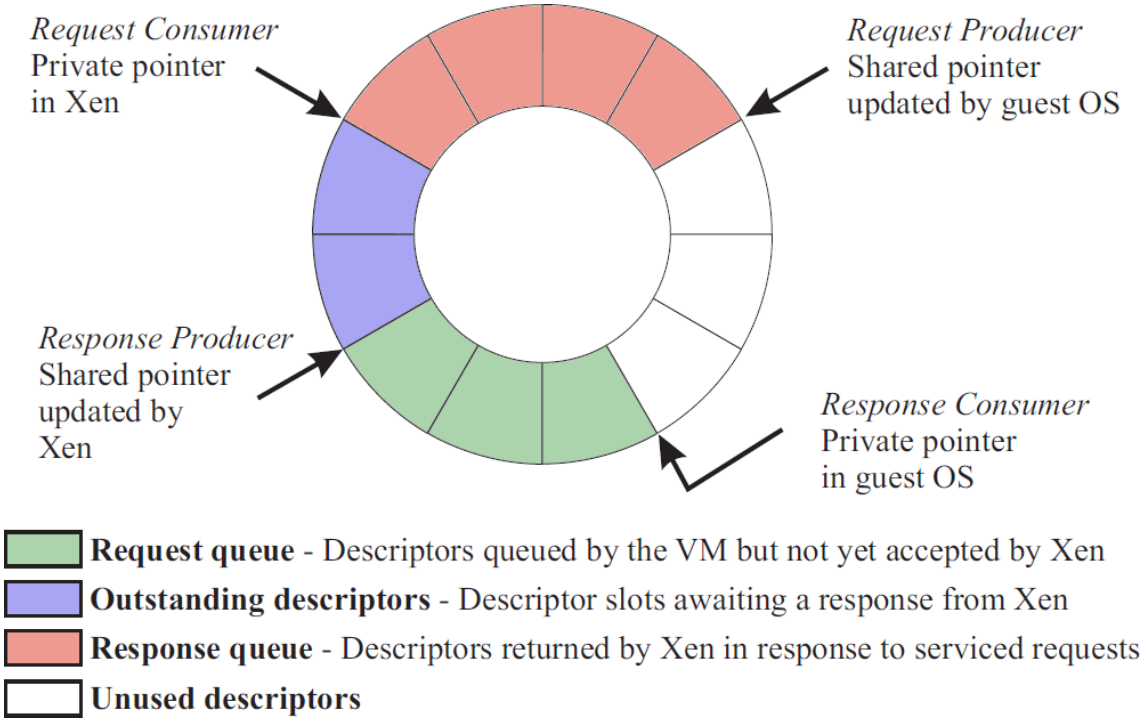


Figure 12 - Structure of the asynchronous I/O ring [4, p 6]

3.1.4.3 Reception

The incoming interrupts from the physical NIC are handled by the Hypervisor. The hypervisor transforms the interrupt to virtual interrupt (event), and forwards it to the Netback driver via the event channel. When the Domain0 is scheduled, its driver will check that there are any forwardable messages. If there is, it forwards the packet to the destination domain. First, the Netback driver updates the reception I/O ring of the reception domain, after it notifies the event channel of the destination domain. When the reception becomes scheduled, its Netfront driver notices the event on the event channel and processes the message from the I/O ring. The following figure illustrates how the message reception is working.

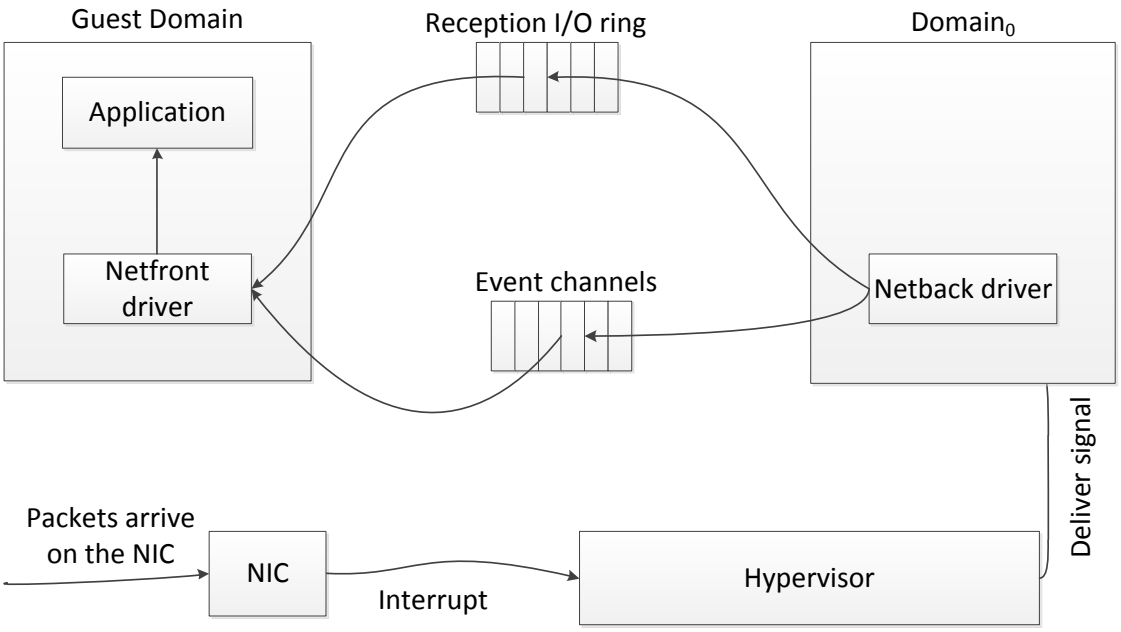


Figure 13 - Network package reception in Xen [16, p .4]

3.1.4.4 Transmission

The transmission of a network package of a guest domain works very similar to that of the reception. The transmission guest domain updates its I/O ring and notifies the Domain0 over the event channel. When the Domain0 becomes scheduled, its Netback driver notices the deliverable network package and sends it directly to the physical NIC. This time there is no need to notify the hypervisor, which accelerates the communication. The following figure illustrates how the network package transmission is working.

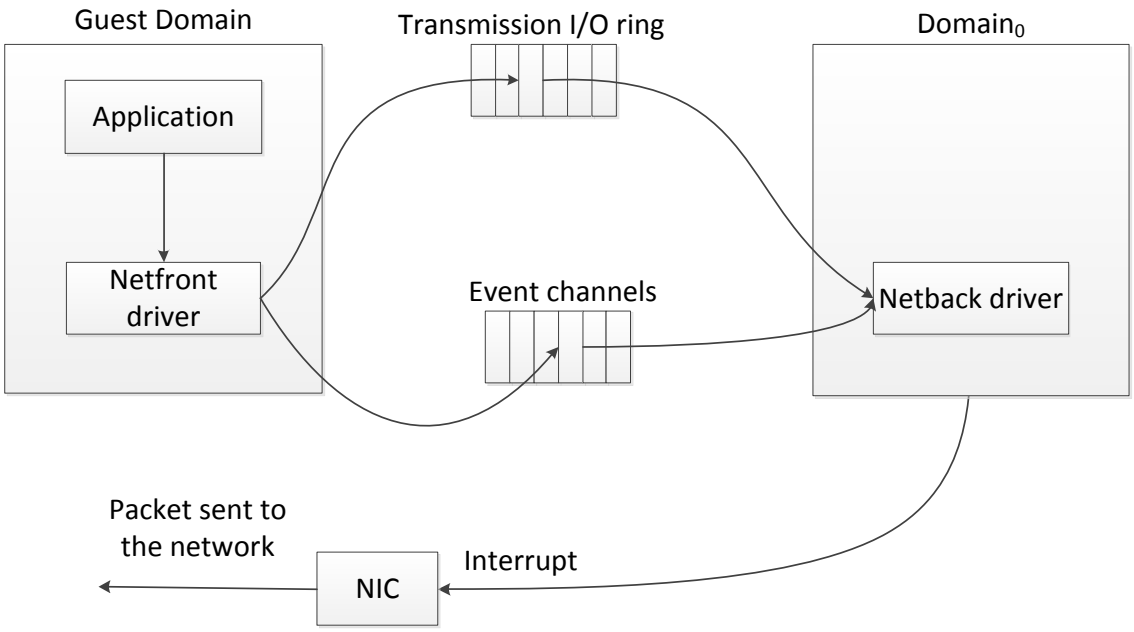


Figure 14 - Network package transmission in Xen [16, p .4]

3.1.5 Schedulers

3.1.5.1 Definitions

A **non-preemptive** scheduler picks a VM for a predetermined time and allows it running till the end of its time slice. The scheduler is not able to interrupt the execution of the VM. The VM is only be able to stop early, if it gives up the processor or became blocked. A **preemptive** scheduler rechecks the scheduling decision time to time. If the priority of a runnable VM becomes higher than the priority of the running VM, then the scheduler pre-empts (deschedules) the actual VM and schedule the VM with the higher priority.

A VM of a hypervisor - which have scheduler with supporting **work-conserving** mode - can utilize the disengaged CPU time, if another VM was blocked. This ensures that the CPU will run with 100% utilization. In contrary a scheduler which supports only the **non work-conserving** mode, the CPU time of a VM cannot be shared and used by an another VM. It means if there are two VM shares, the underlying physical machine and one of them become blocked, then the other VM cannot utilize the released CPU time.

3.1.5.2 Built in Schedulers

Usually, every Xen release supports a few number of VM schedulers. The administrator of the hypervisor can freely exchange them.

- **Round Robin (RR)**: This scheduler was never used in a real environment. It gives a simple example for those who like to learn and try the functionalities of a basic scheduler.
- **Borrowed Virtual Time (BVT)**: This scheduler no longer used in the current l version of Xen. It worked with low overhead both in single and multiprocessors environment. It supports well the I/O intensive applications. The scheduler has many parameters, which needed to be set carefully by an expert to achieve the optimal performance. The scheduler is preemptive and only supports work-conserving mode, which limits its usability.

- **Simply Earliest Deadline First (SEDF)**: This is the oldest amongst the current schedulers of Xen and it was the default before the Credit scheduler appeared. Each Dom_i domain has three parameters ($slice_i$, $period_i$, $extra_i$). The $slice_i$ tells that how many CPU time needed in every $period_i$ for the actual Dom_i domain. The scheduler then schedules the Dom_i (VCPU) which has the earliest deadline. If the $domain_i$ can get extra CPU time in addition to the required time, the $extra_i$ will tell it. The scheduler distributes the remaining usable CPU time fairly between the domains.
- **Credit Scheduler**: This is the default scheduler in the actual version of the Xen hypervisor. It has very good load balance handling in a multi-processor environment. Each Dom_i has two parameters ($weight_i$, cap_i). The share of CPU time received by the domain is determined by the weight. The default weight is 256. If a domain gets 512 as value for the weight, it gets two times more CPU time then a domain with default weight. The cap_i tells how many percentage of extra time a Dom_i can get. If the value of the entire cap is zero, then the scheduler runs in work-conserving mode. In this way the Credit scheduler is able to operate in work-conserving and non work-conserving mode too. The scheduler associates a credit value to each VCPU. When a VCPU becomes scheduled, it consumes its credits. To maintain the fairness, the scheduler classifies the VCPUs into two groups. If a VCPU is OVER scheduled, then it exhausts its available credits. An UNDER scheduled VCPU ran less than it should. The runnable VCPUs are stored in the run queue, ordered by their class type. The scheduler always choses the first UNDER scheduled VCPU from the run queue.

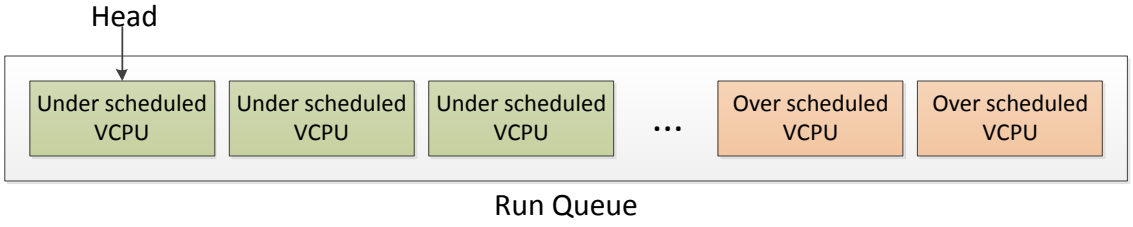


Figure 15 - Run Queue of the Credit Scheduler

The Credit Scheduler has already implemented a type of Co-Scheduling algorithm. The existing two states have been complemented with a new state. The state of a VCPU becomes BOOST, if it is waiting, or it received an I/O message and its state is UNDER scheduled.

- **Credit2 Scheduler**: The Credit scheduler has become 6 years old in 2012, and during these years, technology changed a lot. This is the next generation of the Credit Scheduler, - which is still in experiment state [8] – which gives solution for the weaknesses of the Credit scheduler. The goal of the improved scheduler is to gain better load balance handling and to take advantage of the increased number of cores and the available hyperthreading technique in the CPUs. The work of George W. Dunlap [14] describes the problem of the old Credit scheduler in more detailed way and the study presents solutions for the known problems.
- **ARINC 653 Scheduler**: A 3rd party scheduler, but it is part of the actual schedulers offered by Xen. It implements the ARINC 653 standard compatible CPU scheduler. Steven H. VanderLeest describes the steps of the implementation in his work [28].

4 IMPLEMENTATION

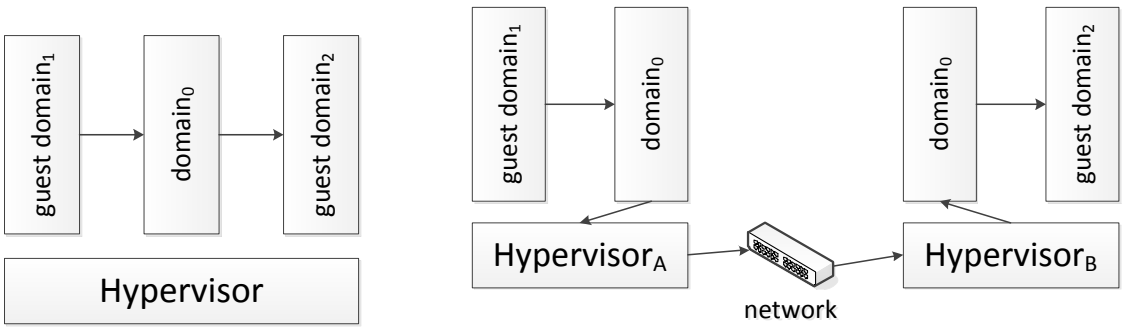
4.1 Choosing appropriate co-scheduling algorithm

The Literature Review chapter of this thesis presents the co-scheduling algorithm already well known in the literature. Nagar et al. [25] highlights a number of co-scheduling algorithms including; the Dynamic Coscheduling, the Periodic Boost and Periodic Boost with Timestamp co-scheduling algorithms.

For this study, the Periodic Boost and Periodic Boost with Timestamp algorithms have been chosen, because they fit better into to the scheduling environment of Xen. The Dynamic Coscheduling algorithm was not selected as it has high overheads.

4.2 Barrier of co-scheduling

Two communicating VMs are successfully co-scheduled, if they are running in parallel in a part of their execution time. The communication part of the Background chapter describes how the packet sending mechanism is working between two domains. The domain0 should be scheduled for each occasion of communication between guest domains. VMs cannot be co-scheduled on physical machines which have only one CPU under Xen. It is apparent on the Figure 16(a) and on the Figure 16(b) that the domain0 should be scheduled for the successful communication and guest $domain_1$ and guest $domain_2$ cannot run in parallel. In case of a) 3 CPUs necessary for co-scheduling, while in case b) 2 CPUs are enough.



a) guest on the same VMM b) the two guests on different VMMs

Figure 16 - Communication between guest domains

Some parts of the work of Sriram Govindan et al. [15, 16] are parallel with this thesis. The aim of their study is different. They intended to decrease the communication time of the VMs. Their solution for the problem is to schedule the communicating domains earlier. In this way, the time of the communication is decreasing. It increases the possibility that the guest domains involved in the communication are running in the same time, thus they solution is a type of co-scheduling algorithm.

4.2.1 Sources of delays

If two guest domains are communicating with each other, then the communication can suffer from various types of delays. The Figure 17 illustrates the sources of delays.

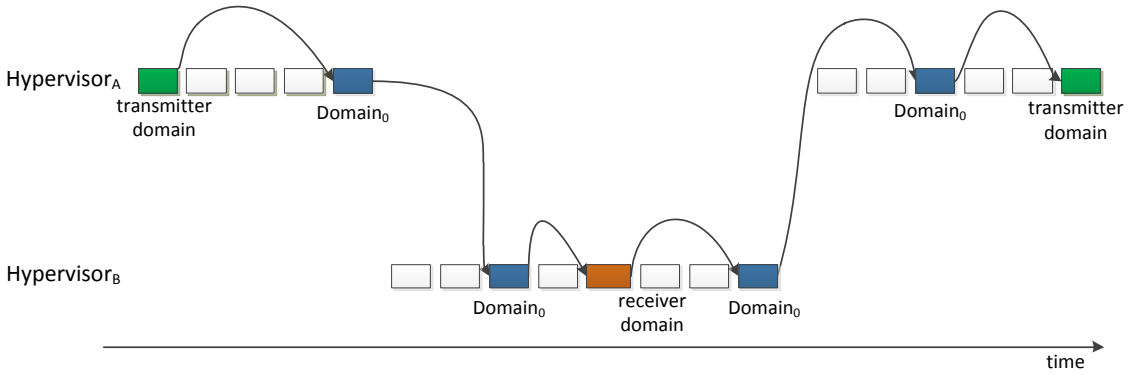


Figure 17 - Sources of delays

Six types of delays can appear during the communication period. 1) The first delay can appear when other guest domains are scheduled after the transmitter domain, thus the message arrives late to domain0. 2) The second delay can be occurred, when the hypervisor sends a message to the other hypervisor and the domain0 of the other VMM is not scheduled when the message arrives. 3) The next type of delay is when the domain0 of the receiver hypervisor sends a message to the receiver domain and other domains become scheduled before the receiver domain. Three final three types of delay are linked to answering the message and are similar to the previous three cases. If the scheduler can increase these delays then the possibility of co-scheduled running of the VMs is increasing.

4.3 Co-scheduling based on observed communication activity

4.3.1 Chosen algorithms to implement

The following versions have been chosen to be implemented:

1) Periodic Boost
    a) Chooses for scheduling the guest domain, which has the highest number of pending messages.
    b) Chooses domain0, if it has pending messages.
    c) Chooses the domain which has the highest number of pending messages. (merge of 1.a and 1.b)
2) Periodic Boost with Timestamp
    a) Chooses the domain which is able to receive the “most recently pending message”. If there is no domain like this, then the normal SEDF behaviour continues.
    b) Chooses the domain which is able to receive the “most recently pending message” in the last n (e.g. 20) msec.
        i) If there is no domain like this, then the normal SEDF behaviour continues.
        ii) If there is no domain like this, then use the 1.c) scheduler.

To implement this scheduler algorithm, the scheduler needs to know that how many pending messages each domain have, and when the messages have arrived.

4.3.2 Creating communication based statistics

The created communication are based on the statistics provided by Sriram Govindan et al. [15, 16]. The statistics are maintaining the number of pending messages and the time of the most recently arrived pending message of each domain. It needs several modifications in the following areas:

- Netback driver of the kernel of the domain0’s OS
- Netfront driver of the kernel of the guest domain’s OS
- Event channel and the scheduler of the Hypervisor.

Every domain has a special memory page, the Shared Info Page, which is stored in the memory of the Hypervisor. It contains runtime information about the system. It was possible in the 3.0 version of Xen, to extend this page and store extra runtime information. Each guest domain is able to store its temporary statistics in the Shared Info Page. When the scheduler of the Hypervisor runs, it is able to collect the temporary statistics of the previously running guest domain, and update and store the overall statistics in the Shared Info Page domain0. The following figure shows that which source files need to be modified to achieve the bookkeeping statistics.

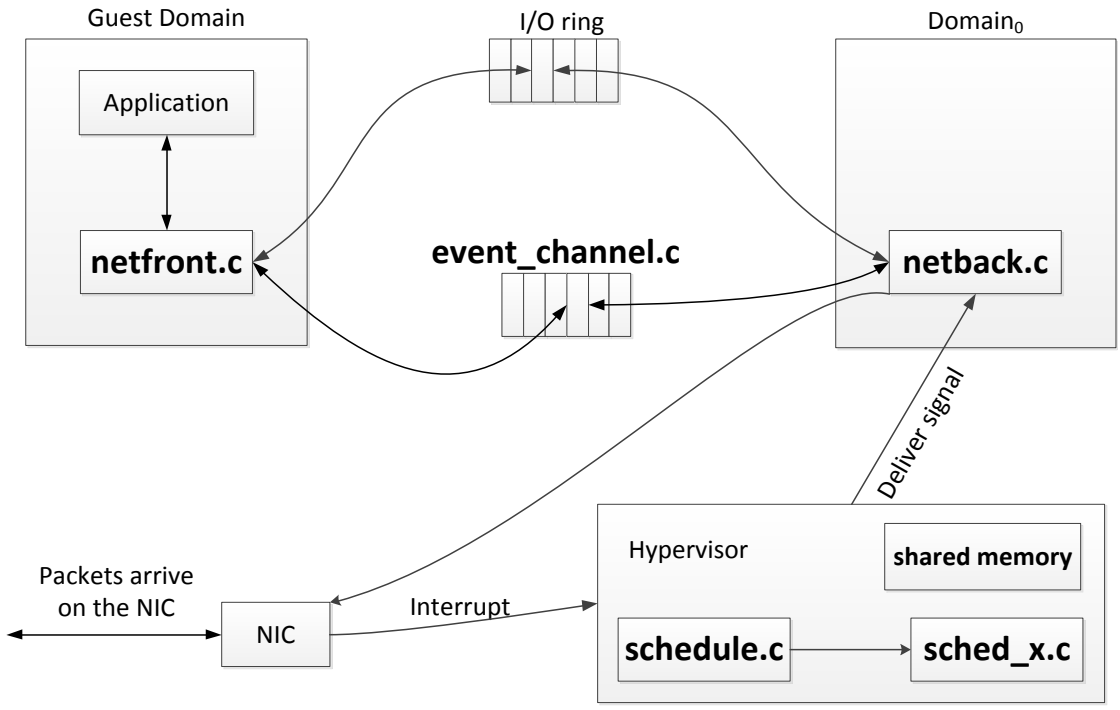


Figure 18 - Implementing communication based statistics

The modifications have been made on the source files:

- **xen.h**: This file needed to be modified both in the kernel and in the hypervisor as well, in order the keep the consistency. It contains the definition of the Shared Info Page, which is shared between the domains and the hypervisor. The modification extends the `shared_info` struct with the following fields:

```
#define MAX_DOMAINS 32

typedef struct shared_info {
      ...

      unsigned long net_ intensity[MAX_DOMAINS];
      uint64_t time_of_last_ intensity[MAX_DOMAINS];
      int number_of_pending_messages_left;

} shared_info_t;
```

Figure 19 - Modification of the Shared Info Page

The `net_ intensity` array stores the amount of pending messages for each domain, while the `time_of_last_intensity` stores the time while the last received pending network package for each domain. Each guest domain informs the hypervisor about how many packages were left to process after its run with the `number_of_pending_messages_left` variable.

- **event_channel.c**: Contains the collection of event channel handling functions in the hypervisor. If an incoming network package goes to the domain0 from the physical NIC, it also goes through the event channel of domain0. The modification affects the `send_guest_pirq()` function. If an incoming message goes to the domain0, then the function increases the `net_intensity[0]` - namely the number of pending messages of domain0 - by one and assign the actual time to the `time_of_last_intensity[0]` in the bookkeeping page of domain0.
- **netback.c**: This driver responsible for the communication and part of the kernel of domain0. If one of the guest domains receives a message, it goes through the domain0, which forwards the package to the appropriate domain. The modification increases the pending network packages of the guest domain (`net_intensity[i]++`) and refreshes the actual time" of the `time_of_last_intensity[i]` variable in the bookkeeping page of domain0 at the function which is responsible for the package forwarding.

- **netfront.c**: This driver responsible for the communication of the guest domain and part of the kernel. If a guest domain sends a message, then the `network_start_xmit()` is executed. This function increases the amount of pending messages of domain0 by one, and refresh the time value of `time_of_last_ intensity[0]` in the bookkeeping page of the guest domain. Furthermore, when a guest domain will be scheduled, it is possible, that it is not able to process all events. Thus, it is required that the guest domain should know, how many unprocessed events it has in its own bookkeeping page.
- **schedule.c**: The `__enter_scheduler()` function is executed before every scheduler decisions. This function calls the chosen scheduler to select a VCPU to execute. Before this function call, the function merges the Shared Info Pages of the last executed guest domain and the domain0 into the Shared Info Page of domain0. This keeps the bookkeeping up-to-date. When the function receives the chosen VCPU to execute, it resets the values of the domain of the VCPU.

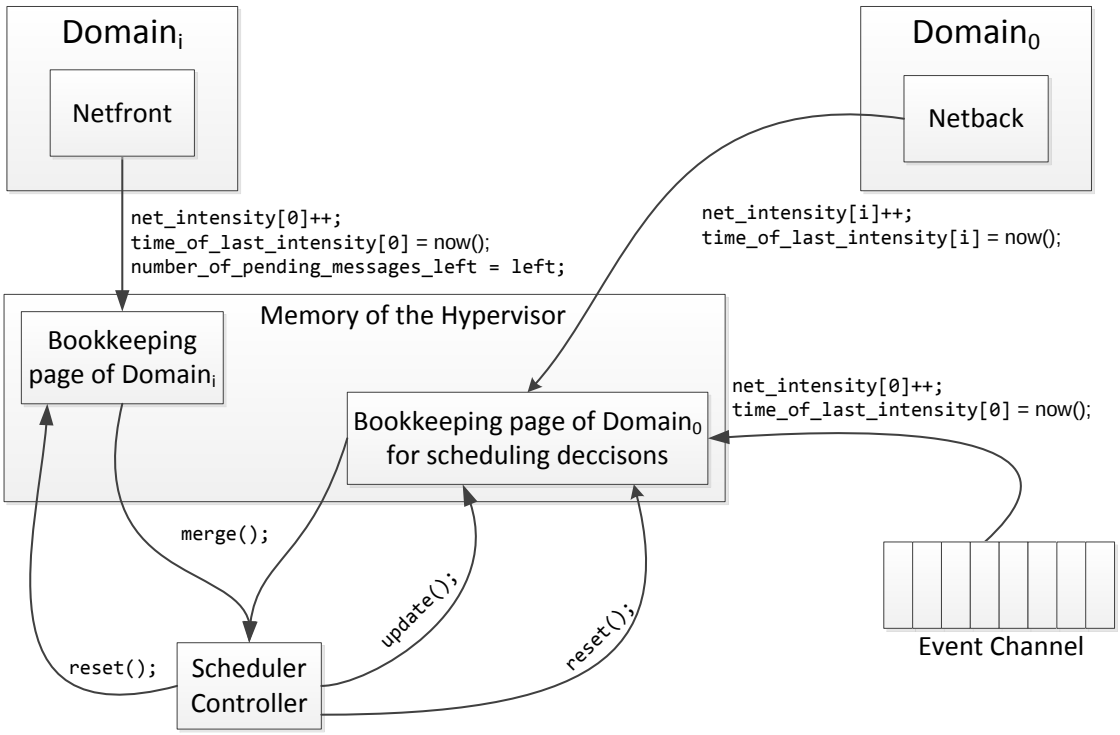


Figure 20 - Implementation of the Statistics of the Communication

- **sched_x.c**: The modified scheduler can make decisions by right of the statistics of the domain0.

4.3.2.1 Deprecated Solution

When I created a prototype of the implemented statistics, I ran into an error. This thesis is made under the most recently version of Xen (v4.1.2). Unfortunately, it is impossible to modify or extend the Shared Info Pages in the actual version of Xen. According to the Xen Developers mailing list [34], its size has been fixed and it cannot be modified, because it has been used for optimization.

4.3.2.2 Bug in the statistics of Communication aware CPU Scheduler

Sriram Govindan et al. [15, 16] made a communication aware scheduler, which was a new approach when it was realised. Their work impressed me and helped me to understand, how the Xen system is working. The source code of their scheduler can be found on their website [33]. I think while I studied their scheduler, I found a bug in their source code.

They distinguish the incoming and the outgoing network activity of domain0. To store these values, Shared Info Page has been extended with two fields, `dom0_tx_intensity` for the transmission intensity of domain0 and `dom0_rx_intensity` for the reception intensity of domain0. Their statistics has been prepared to support further works, so the modification maintains the total amount of incoming and outgoing messages of domain0. However, they are not used by the scheduler.

```
unsigned long dom0_tx_intensity; // transmission intensity
unsigned long dom0_rx_intensity; // reception intensity
...
unsigned long total_dom0_tx_intensity;
unsigned long total_dom0_rx_intensity;
```

Figure 21 - Part of the Share Page of the Communication aware CPU scheduler

The network intensity of domain0 is essential and their modified scheduler makes important decisions about these values (`dom0_tx_intensity`, `dom0_rx_intensity`).

Nevertheless, the value of `dom0_tx_intensity` is never changes, it always stays 0. The value should be updated in the netfront driver of the guest kernel, when the guest sends a network package to the domain0, but this does not happen. It only increases the `total_dom0_tx_intensity` field of the Shared Page.

There is only one place in the source code, where the authors try to increase the value of `dom0_tx_intensity`. It is in the controller part of the scheduler (`scheduler.c`). The value of the `dom0_tx_intensity` in the Shared Page of domain0 is increased by the value of `dom0_tx_intensity` from the Shared Page of the previously ran guest domain, which is zero. In addition, two lines later, the value of the `dom0_tx_intensity` in the Shared Page of domain0 is set to zero. The following source code contains the mentioned rows.

```
/* The network packets that have been transmitted by the domUs
are added to dom0's reception intensity, since those packets has
to be processed by the domain0 and send to the NIC card  */
dom0->shared_info->net_rx_intense[prev->domain->domain_id]+=
     prev->domain->shared_info->dom0_tx_intensity;
dom0->shared_info->dom0_tx_intensity+=
     prev->domain->shared_info->dom0_tx_intensity;
dom0->shared_info->total_dom0_tx_intensity+=
     prev->domain->shared_info->dom0_tx_intensity;
dom0->shared_info->dom0_tx_intensity=0;
prev->domain->shared_info->dom0_tx_intensity=0;
```

Figure 22 - Part of the source code of the Communication aware CPU scheduler

4.4 Co-scheduling based on built-in communication statistic

4.4.1 Hierarchy of the Shared Info Page

The shared info page is a shared memory between the guest and the hypervisor. It contains lots of dynamically changing information from the global system.

The shared info page is a C language struct, which contains few more nested struct structures, as it is apparent on Figure 23. It contains actual information of the VCPUs, the status of the event channels and more architecture dependent descriptions.

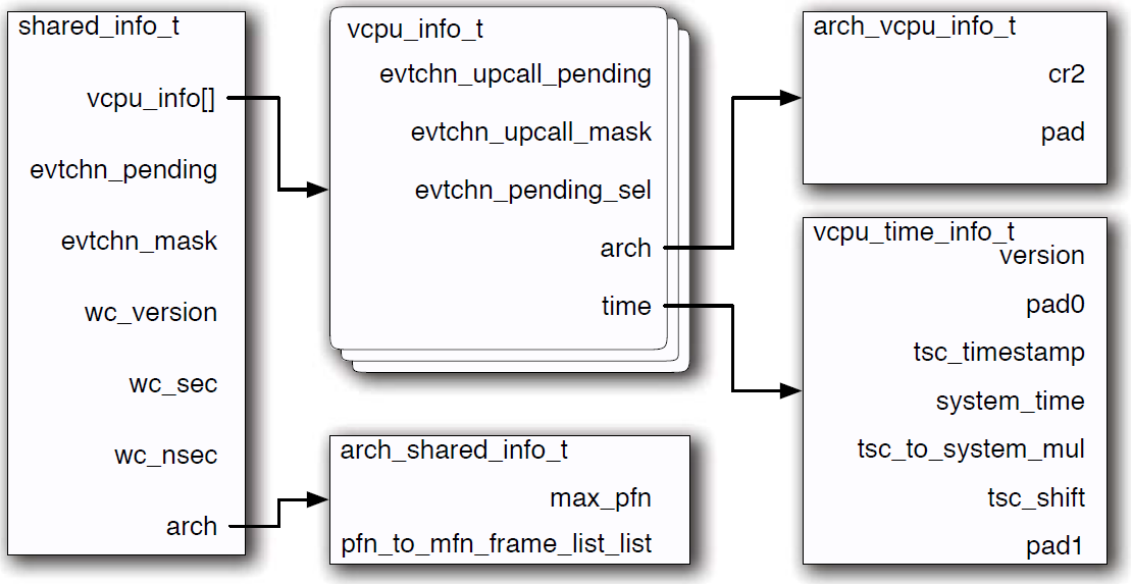


Figure 23 - Hierarchy of the Shared Info Page [9, p. 51]

The first field, the `vcpu_info` is an array, which contains the VCPUs assigned to the actual domain. Each `vcpu_info_t` contains virtual event handling, time and architecture based information.

Domains are able to create event channels, these are used to receive and send event notifications. The shared page contains the `evtchn_pending` field as well, which is used to notify the domain about a pending notification. This field is 8 machine words long. Thus, on a 32 bit system, a guest domain can obtain 256 event channels. The notification is set by the hypervisor, while the guest resets

the value after it processed the event. The `evtchn_mask` describes the delivery mode, which can be asynchronous or synchronous. The appropriate bit of `evtchn_pending` has been set every time when an event generated. The following source code is part of the `event_channel.h` and describes the possible channel states. The most important state is the EVTCHNSTAT_virq, which is used for the communication between domains.

```
#define EVTCHNSTAT_closed      0 /* Channel is not in use.                */
#define EVTCHNSTAT_unbound     1 /* Channel is waiting interdom connection.*/
#define EVTCHNSTAT_interdomain 2 /* Channel is connected to remote domain. */
#define EVTCHNSTAT_pirq        3 /* Channel is bound to a phys IRQ line.   */
#define EVTCHNSTAT_virq        4 /* Channel is bound to a virtual IRQ line */
#define EVTCHNSTAT_ipi         5 /* Channel is bound to a virtual IPI line */
```

Figure 24 - Possible states of the event channels

4.4.2 Implementing Scheduler

Xen offers an abstract interface to create new schedulers. The approach is very similar to OO programming.

Each scheduler needs to define the name, the option name and the ID of the scheduler. Only the `do_schedule()` method is mandatory to be implemented, all other methods are optional. The following table contains the major methods.

Method	Description
`do_schedule`	Responsible to select the next VCPU to schedule.
`wake`	The scheduler receives a task to run, which will be inserted into the RunQ.
`pick_cpu`	Rendering the VCPU to physical CPU.
`sleep`	Called when the VM is paused.

Table 3 - Some of the major methods of the schedulers

The following struct definition describes the structure of the scheduler's interface.

```
struct scheduler {
  char *name;              /* full name for this scheduler      */
  char *opt_name;          /* option name for this scheduler    */
  unsigned int sched_id;   /* ID for this scheduler             */
  void *sched_data;        /* global data pointer               */

  int   (*init)           (struct scheduler *);
  void  (*deinit)         (const struct scheduler *);
  void  (*free_vdata)     (const struct scheduler *, void *);
  void  (*alloc_vdata)    (const struct scheduler *, struct vcpu *,
                           void *);
  void  (*free_pdata)     (const struct scheduler *, void *, int);
  void  (*alloc_pdata)    (const struct scheduler *, int);
  void  (*free_domdata)   (const struct scheduler *, void *);
  void  (*alloc_domdata)  (const struct scheduler *, struct domain *);
  int   (*init_domain)    (const struct scheduler *, struct domain *);
  void  (*destroy_domain) (const struct scheduler *, struct domain *);
  /* Activate / deactivate vcpus in a cpu pool */
  void  (*insert_vcpu)    (const struct scheduler *, struct vcpu *);
  void  (*remove_vcpu)    (const struct scheduler *, struct vcpu *);
  void  (*sleep)          (const struct scheduler *, struct vcpu *);
  void  (*wake)           (const struct scheduler *, struct vcpu *);
  void  (*yield)          (const struct scheduler *, struct vcpu *);
  void  (*context_saved)  (const struct scheduler *, struct vcpu *);
  struct task_slice (*do_schedule) (const struct scheduler *,
                             s_time_t, bool_t tasklet_work_scheduled);
  int   (*pick_cpu)       (const struct scheduler *, struct vcpu *);
  void  (*migrate)        (const struct scheduler *, struct vcpu *,
                           unsigned int);
  int   (*adjust)         (const struct scheduler *, struct domain *,
                           struct xen_domctl_scheduler_op *);
  int   (*adjust_global)  (const struct scheduler *,
                           struct xen_sysctl_scheduler_op *);
  void  (*dump_settings)  (const struct scheduler *);
  void  (*dump_cpu_state) (const struct scheduler *, int);
  void  (*tick_suspend)    (const struct scheduler *, unsigned int);
  void  (*tick_resume)     (const struct scheduler *, unsigned int);
};
```

Figure 25 - Interface of the schedulers

There are additional methods for the purpose to create and destroy VCPUs and domains and to support dumping and logging activity.

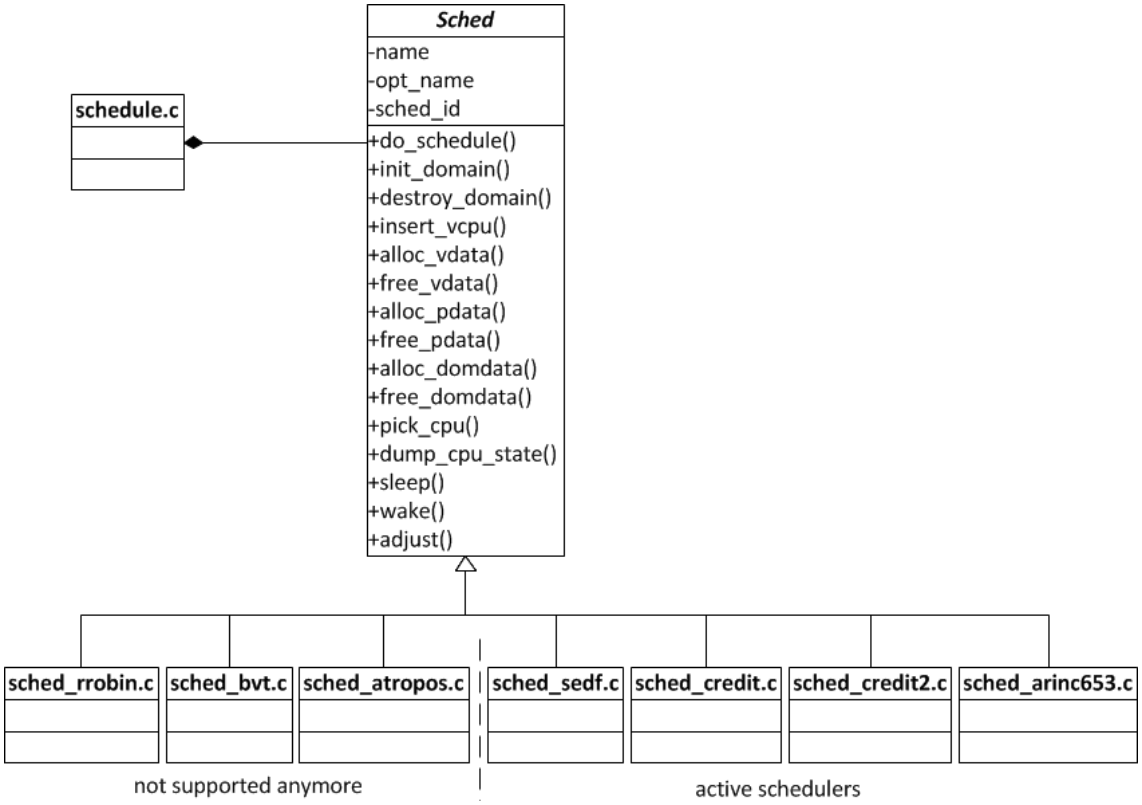


Figure 26 - OO design approach of implementing schedulers

The `scheduler.c` contains all of the scheduler independent implantations and the controller of the actual scheduler. It will call the functions of the scheduler, such as asking which VCPU will be the next to be scheduled. If a method of the interface in not implemented, - namely defined with NULL - then the controller simple ignores the function call.

4.4.3 Choosing built-in scheduler for the modification

The two most widely used schedulers of the Xen are the SEDF and the Credit. Credit scheduler performs better in multi-processor environment, but is more complex as well. We chose the SEDF to modify for the experiment, because it is easier to understand its way of operation.

4.4.4 Implementation details

4.4.4.1 Algorithms

Two algorithms have been implemented. The algorithms prefer the domain which has the most pending messages.

- **CSDD** (Co-Scheduling with Domain Distinguishing): It chooses the domain which has the most pending messages from the available domains, but the algorithm prefers the domain0. If the domain0 has a message to process, then it chooses that one.
- **CSDE** (Co-Scheduling with Domain Equivalence): It choose the domain which has the most pending messages from the available domains, regardless which domains is that. (CSE is a simplification of CSD).

4.4.4.2 Modification of SEDF

The `get_number_of_messages()`function has been created to evaluate the amount of pending messages for a chosen domain. It goes through the event channels of the domain and counts the pending messages. Its source code can be found in the Appendix.

The `sedf_do_schedule` function in the SEDF scheduler has been modified. It is responsible for choosing the appropriate VCPU to execute.

The modification has been made after the point, where the scheduler chose the first VCPU from the RunQ. The algorithm goes through on the RunQ with a for cycle and looks for the VCPU which has the maximum amount of pending messages. There is some difference between the CSD and the CSE. If the domain0 is in the RunQ and it has some pending messages, then the iteration will be aborted and the scheduler chooses the domain0. The code of the CSD can be found in the Appendix.

There is a special function in the SEDF scheduler which name is `sedf_do_extra_schedule()`. If there is no more VCPU, which has deadline in the actual period, then this function is executed to select a VCPU to ensure some extra time for it. Similar modifications have been executed in the `get_number_of_messages()` function.

5 EXPERIMENT

5.1 Experimental Setup

The experimental setup contains two Xen hosted servers. Each server has Core 2 Quad 2.5 GHz CPUs with 3MB of cache, 1333 MHz FSB speed and 4 GB RAM. The machines are connected with Gigabit Ethernet. Xen 4.1.2 and Ubuntu Server 12.04 were installed to each computer following the installation guide of Xen 4.1.2 [30]. Ubuntu 11.04 has been installed to each VM with 128Mb RAM. This small RAM size has been chosen because one of the originally planned test scenarios contains up to 24 VMs, and these need to share the 4GB physical RAM. Although as it turned out later, there was not enough free IP address in the network to run as many VMs in the same time. Furthermore each domain has only one VCPU and each VCPU has been pinned to one physical CPU.

A separated machine has been selected to the role of the main node. It starts and distributes the tests between the VMs. MPI environment has been installed both to the main node and to the VMs. Therefore the VMs are able to execute the NPB tests.

The following naming convention has been used to identify the location of a VM in the system: host<machine_ID><CPU_ID><row_ID>.

There were 18 free IP addresses in the network, so the test cases should be chosen carefully in order to make conclusions from the experiment, but the amount of VMs does not exceed 18.

There were 4 cores in both of the physical machines. Due to the strong number of VMs limitations, we decided to use only 2 cores.

5.2 Chosen Benchmark Programs

The empirical result of Mark Lee Stillwell et al. [19] showed that some of the NPB tests has huge memory requirement. Nevertheless, as the number of VMs increasing in the benchmark, the needed memory size is usually decreasing. The EP benchmark program has the least memory requirements, - less than 64

Mb - independently from the number of VMs. On the other side, the FT has the greatest memory usage. Besides, the LU program is almost the only benchmark program, which needs less than 128 Mb when it runs on 4 VMs. In the study of Shailabh Nagar et al. [25], the authors classified the network communications of the LU and the EP NPB programs. They experienced that there is almost no communication in EP, while LU has a lot of small messages and its communication intensity is roughly 16% of its runtime.

Due to the previously mentioned reasons, the EP and the LU benchmark programs have been chosen to serve as the default building blocks of the test cases.

The following test has been made to count the LU and the EP running time on 4 VMs, in order to help to choose the appropriate benchmark classes for each test case. The VMs were running on separate CPUs, in order to avoid interaction/impact between VMs. The figure below shows the architecture of the created test bed.

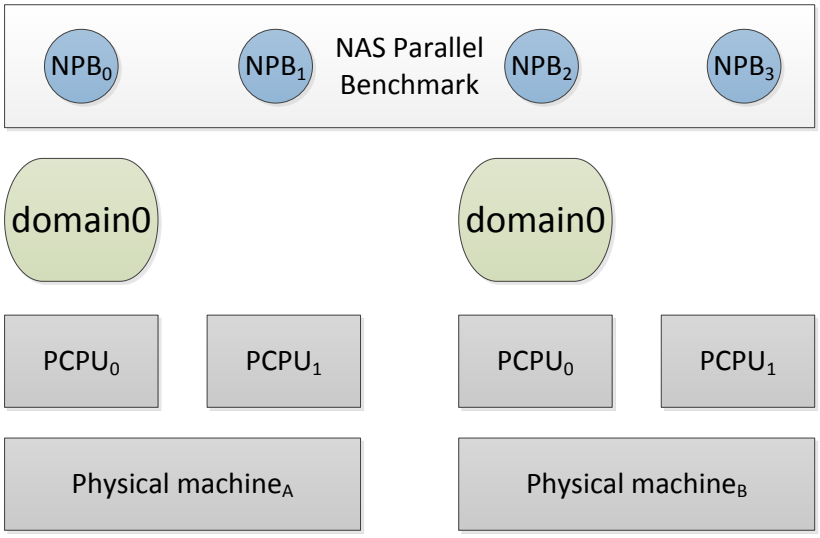


Figure 27 - Test architecture to identify running times of NPB applications

The result of the tests is shown on the following table. It is used as a baseline to help to identify the proper class for the benchmark programs. It is appropriate that the order of the classes is different. Probably, it is a good idea to run an LU application with an EP with a higher level class.

	Class A	Class B	Class C	Class D
Embarrassingly Parallel	6.82	27.35	109.48	1750.05
LU factorization	40.34	225.03	905.34	estimated[1]

Table 4 – The runtimes (in seconds) of LU and EP applications with variant classes

5.3 Execution of the Experiment

The SEDF scheduler has been chosen as a baseline and it will be compared with the two co-scheduler. Each measurement has been executed 5 times and their average value has been calculated, in order to reach more representative results.

The experiment consists of three main test cases. Each test case contains the purpose, the architecture, the expected result and the measured result of the test case.

5.3.1 Overhead and slowdown experiment

5.3.1.1 Overhead of CPU consuming applications

A test case with four sub test cases has been set-up to diagnose the behaviour of the schedulers, when it executes CPU consuming applications. Firstly, only one application runs without parallel execution. The amount of applications is increasing in each test case, until it reaches four EP applications running in parallel. This limit has been determined by the available IP numbers. The test bed is shown in Figure 28.

[1] The estimated value is 16 times more than the value of the previous class.

Probably the scheduler can't produce any speed-up, due to the lack of communication in this application. Therefore, this test case is appropriate to identify the overhead of the new schedulers.

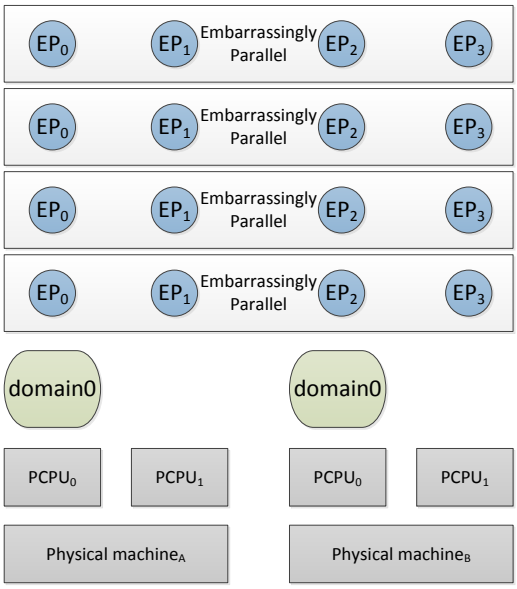


Figure 28 - CPU consuming applications running in parallel

The C class of EP has been chosen to execute the test cases. The following table shows the result.

	SEDF	SEDF-CSDE	SEDF-CSDD
No competition	109.48	109.49	110.95
2 parallel applications	224.44	226.23	225.33
3 parallel applications	337.79	349.12	350.47
4 parallel applications	453.54	485.76	487.81

Table 5 - The runtimes (in seconds) of the CPU consuming applications which run in parallel.

It is apparent, that the scheduler can't produce any speed-ups. There is almost no difference between applications not running in parallel and two applications running in parallel. As the amount of parallel applications is increasing, the two

co-scheudling schedulers are producing more and more overheads. To quantify the number of overhead, the following equation has been created.

$$O_{p,sch} = T_{p,sch} - (T_{1,SEDF} * p) \quad \textbf{(5-1)}$$

Where *p* is the number of parallel applications, *T* is the time of the execution and *sch* is the chosen scheduler. The $O_{p,sch}$ identifies how much longer it takes to execute p parallel applications with the actual scheduler, than executing an application without competition and scheduled by the SEDF scheduler. The evaluation of $O_{p,sch}$ for the previous table is shown in Figure 29.

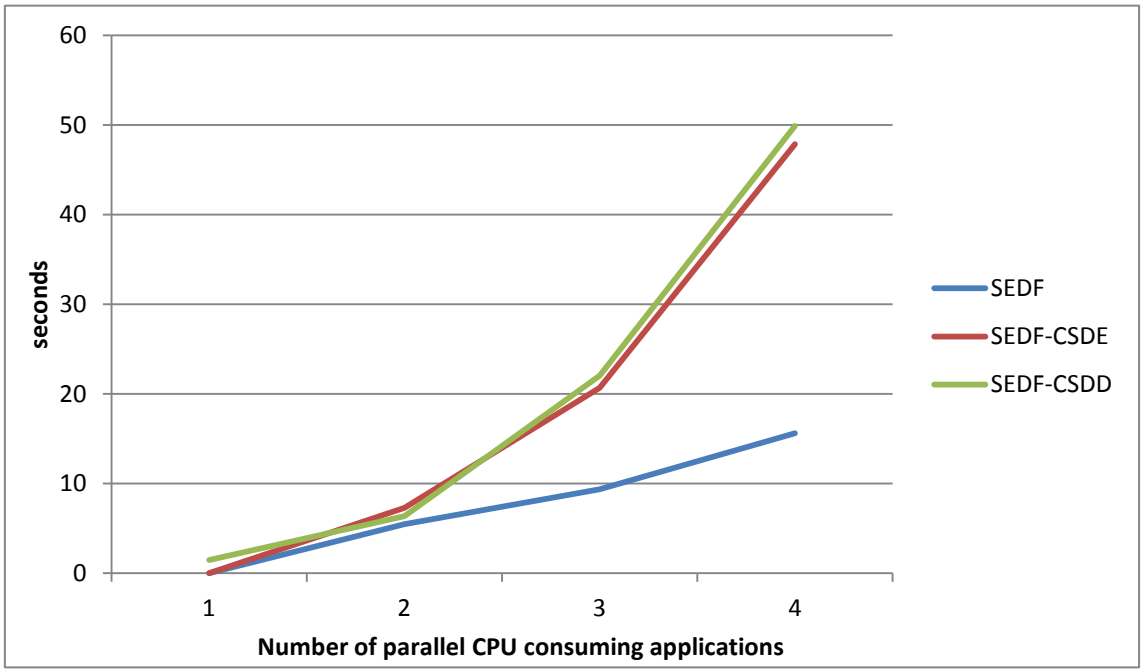


Figure 29 - Overhead of scheduler when executing CPU consuming applications

The overhead of the two new schedulers are significant. However it is not surprising. When the co-scheduling schedulers decide to choose a VCPU - which will run on a CPU, - it checks the RunQ – which contains the runnable and scheduled VCPUs, - and chooses the VCPU, which has the most pending messages on the event channel. If the amount of VCPUs in the RunQ is N and the length of the array of the event channel is M, then the time complexity of the scheduler algorithms is $O(N * M)$.

5.3.1.2 Slowdown of CPU consuming applications

The following test analyses the case when only network insensitive applications are running in parallel. Previously it was mentioned, that the LU application has been chosen for this type of test cases, because it has high network I/O intensity. A test bed similar to the previous one has been created for the CPU intensive applications, as it is shown in Figure 30.

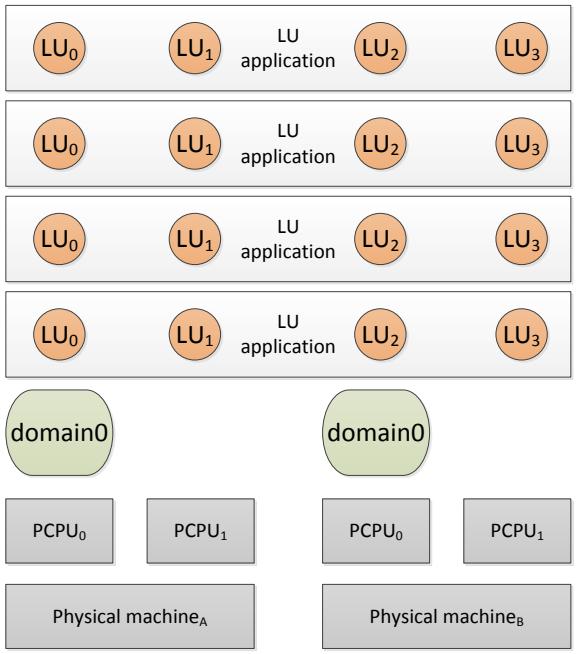


Figure 30 - Network I/O intensive applications running in parallel

The following table contains the execution times of parallel running LU applications with class B.

	SEDF	SEDF-CSDE	SEDF-CSDD
No competition	225.03	225.25	225.37
2 parallel applications	448.08	453.02	452.87
3 parallel applications	747.62	704.56	708.89
4 parallel applications	1005.72	977.01	980.31

Table 6 - The runtimes (in seconds) of the network I/O intensive applications which run in parallel.

If there is no competition for an application, then the performance of the three schedulers are quite similar. When two I/O intensive applications are running together, then the two new schedulers have small overhead. Nevertheless, the co-scheduling schedulers are able to gain significant advantage when 3 or 4 applications are running together. In order to gain better understanding of the speed-up, the following equation for slowdown has been evaluated for each element of the table.

$$S_{p,sch} = \frac{T_{p,sch}}{T_{1,SEDF}} \quad \textbf{(5-2)}$$

This equation defines the slowdown, namely how much the actual execution runs slower than the application without competition with SEDF scheduler. The SEDF and the SEDF-CSDE has been highlighted from the result and the interval between 2 and 4 has been enlarged.

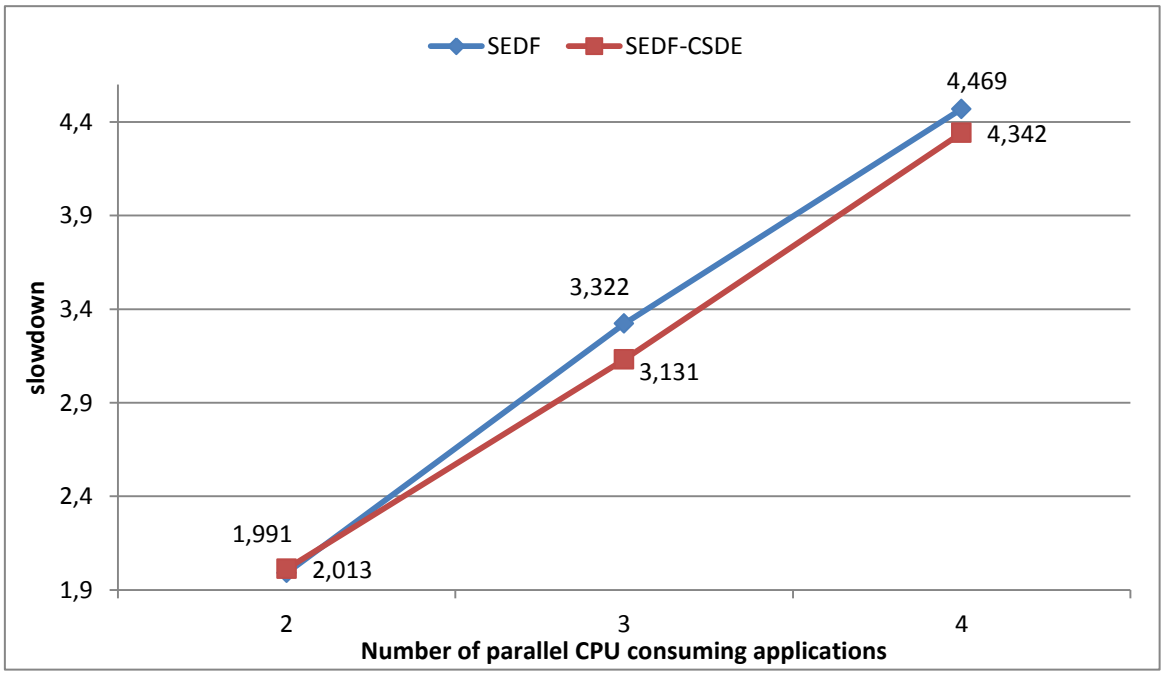


Figure 31 - Slowdown of SEDF and SEDF-CSDE when I/O intensive applications running in parallel

It is difficult to see, but the advantage of the SEDF-CSDE is decreasing. It is caused by the previously detected overhead, i.e. the O(N*M) time complexity of the searching algorithm of the co-scheduling schedulers. Probably when 5 I/O intensive applications are running in parallel, the slowdown of the two highlighted schedulers should be very close to each other.

Another result is that the performance problem of the I/O intensive applications is apparent. The slowdown of the parallel running LU applications and the slowdown of the parallel running EP applications have been compared in Figure 32. The slowdown of the high network intensive application is increasing faster as the amount of parallel applications is increasing. This comparison is not entirely perfect and it is different in each application, but it clearly shows the main problem of the parallel running I/O intensive applications, what this thesis intends to solve.

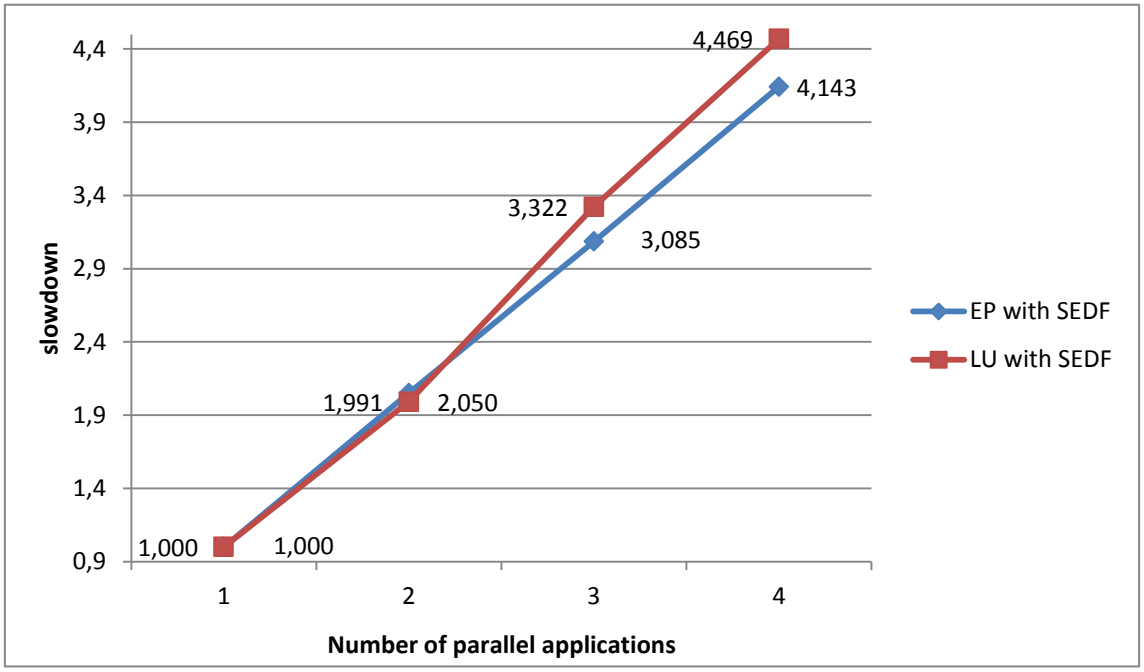


Figure 32 - Slow down of a CPU and a Network intensive application

It is not entirely sure that the two co-scheduling algorithms are able to gain advantage when I/O intensive applications are running in parallel. If every VM of the applications are communicating to the same extent very intensively, then it is possible, that there is no best VCPU to choose. This can occur, because all

VCPUs are communicating the same amount in their period. It could be an interesting benchmark to change the LU algorithm to a more intensively communicating application, - like IS or MG, - but the limited amount of memory did not allow for it.

5.3.2 Optimal test cases

It is an optimal situation for a communication intensive co-scheduled application, when it's running among CPU intensive processes. Probably when a network I/O intensive application is communicating, it will take precedence over the EP application and its VCPUs will be scheduled before the other VCPUs. Thus it is expected that if a co-scheduled LU application runs among EP applications, then the LU application should finish its execution earlier.

The architecture of the default test bed is shown on the following figure.

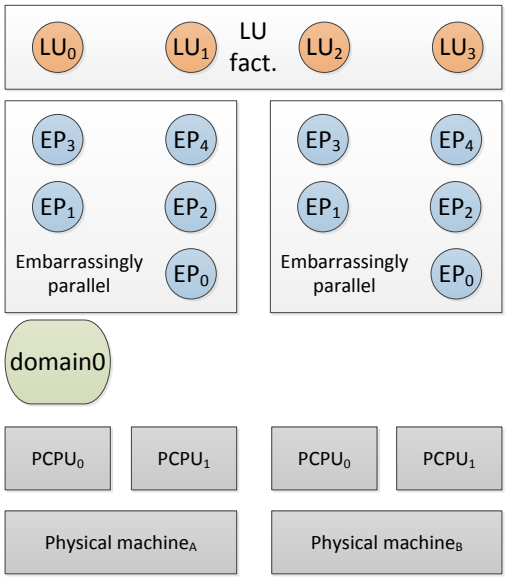


Figure 33 - The architecture of an optimal test case

The class of the EP application have been chosen in such a way that the EP application runs longer than the LU application. This is important, because if the EP finishes earlier, then the LU will run solely. In this case, there is no point of co-scheduling the LU application with itself. The C class of the EP doesn't cover the running time of the LU with B class – with the same amount of VMs. Therefore the D class has been chosen to solve 16 times bigger problem(s)

than the C one. It has high execution time, but it was important to cover the running time of the LU application with class B. The execution times of the mixed test case are shown on the following table.

	SEDF	SEDF-CSDE	SEDF-CSDD
EP (class D)[2]	4496.79	4819.72	4831.26
LU (class B)	717.25	572.86	575.19

Table 7- Execution times (in seconds) of an optimal test case under various schedulers

Both of the two new schedulers prefer the network I/O communicating VCPUs, therefore it is not surprising that the LU finishes earlier when it is running with the co-scheduler. Nevertheless, the running time of the CPU consuming application is effected by the overhead of the scheduler. Consequently, the running time significantly increased.

The LU application finished its run earlier. After this point, the EP gets more CPU time and it can use 100% of efficiency of the CPU. If the overhead of the scheduler would have smaller time complexity, there would be a chance that the EP finishes its running earlier as well.

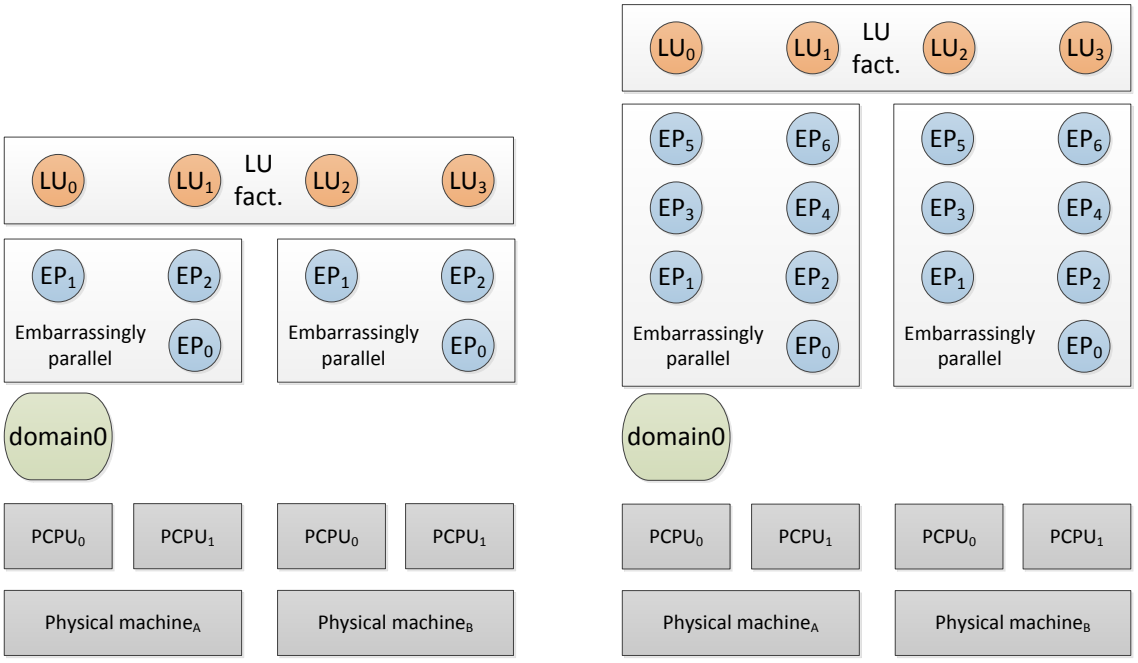
It is apparent, that the domain distinguishing version performs a bit worse than the domain equivalent version. Maybe it caused by the fact, that the $domain_0$ is scheduled more often by the SEDF scheduler than a simple guest domain, so there is no point to handle it in a distinguished way.

It is an interesting fact that the running time of LU under the SEDF scheduler is 717.25 seconds. In the previous test case, when LU applications were being run in parallel, the running time of the LU application was 1005.72 seconds with almost the same amount of running VCPUs. At first sight, the huge difference can be confusing. Nevertheless the article of Angela C. Sodan and Lei Lan [26] and the study of Xing Pu et al. [23] clarify the speed up. The combination of the

[2] average execution time of the two groups

CPU-bound and network-bound VMs results low competition for the resources and the interference between the VMs is minimal.

The next experiments will measure that what is happening when the number of CPU consuming VCPU is changing in the system compared to the previous test case. The architecture of the two changed test cases is visible in the following figure.



a) decreasing the number of VCPUs b) increasing the number of VCPUs

Figure 34 - Modifying the optimal test case with variant number of CPU consuming VCPUs

The result of the test, when the amount of the CPU consuming VCPUs on the physical machines has been decreased to three, is shown on the Table 8.

	SEDF	SEDF-CSDE	SEDF-CSDD
EP (class C)	4893.10	5033.69	5042.81
LU (class B)	529.34	446.80	438.31

Table 8 - Execution times (in seconds) of the optimal test with decreased number of VCPUs

The running time of LU application with the SEDF scheduler is decreased, compared to the previous result (Table 7). It caused by that the application has less VCPUs to compete, thus it runs faster. It is interesting, that the running time of the CPU consuming application is increased with SEDF. It became possible, because at the beginning of the overall running time, the LU application got more CPU time in the ratio of execution time. However when the LU application has been finished, only the three VCPUs of the EP application shared the two physical CPUs, thus the EP should be finished earlier than previously.

The new algorithms behave similarly than in the previous test. In case of EP, they make less overhead, because there is less VCPUs in the system. The LU application runs faster this time too. This is the only test case, where the SEDF-CSDD is a better co-scheduler than the SEDF-CSDE.

The following table contains the results of the extended case, when seven CPU consuming applications are running in 1 physical machine.

	SEDF	SEDF-CSDE	SEDF-CSDD
EP (class C)	4339.43	4951.23	4970.62
LU (Class B)	916.45	732.93	746.99

Table 9 - Execution times (in seconds) of the optimal test with increased number of VCPUs

The overhead on the EP application is the highest amongst the three test cases which were scheduled by the two new schedulers. The running time of the LU application is very high, because it needs to share the resources with many VCPUs. Nevertheless, the co-scheduling schedulers can help on this problem, and they can achieve a significant improvement.

In order to demonstrate the speed up of the LU and the slowdown of the EP applications, two diagrams have been created to summarize the execution times. The first diagram - Figure 1 - shows the execution times of the LU

application with various schedulers and the EP application with various amounts of VCPUs.

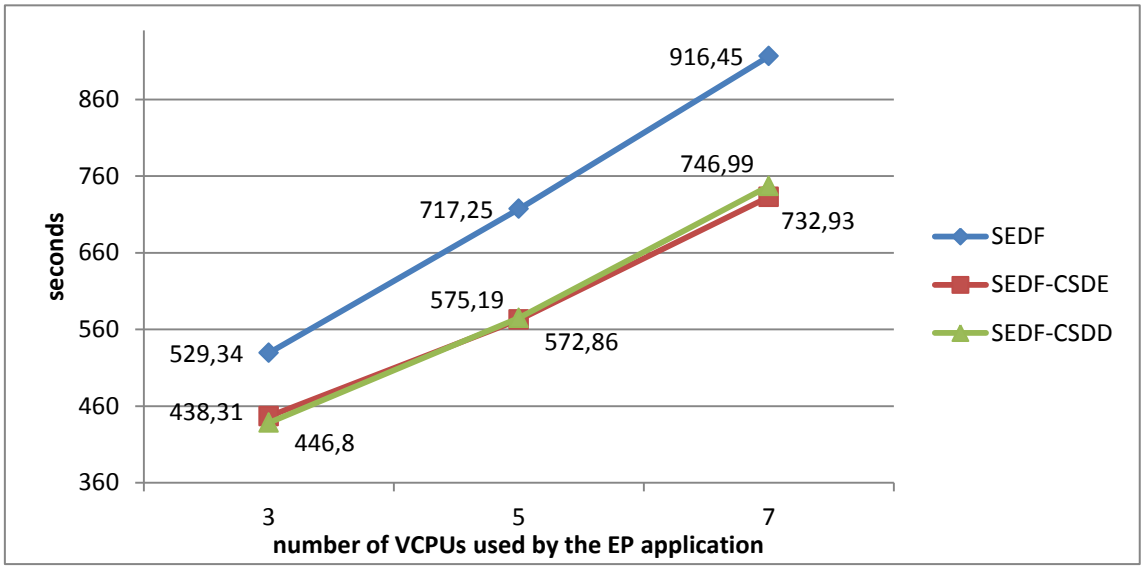


Figure 35 - Execution times for the LU application with various schedulers and various numbers of VCPUs to share the physical CPUs

When the number of VCPUs is increasing, - i.e. the application needs to share the CPUs with more VCPUs - the execution time of the LU application is increasing too. Nevertheless, it seems that as the amount of VCPUs is increasing, the co-scheduler is able to provide better running time for the LU application. The following diagram shows the same situation with the EP application.

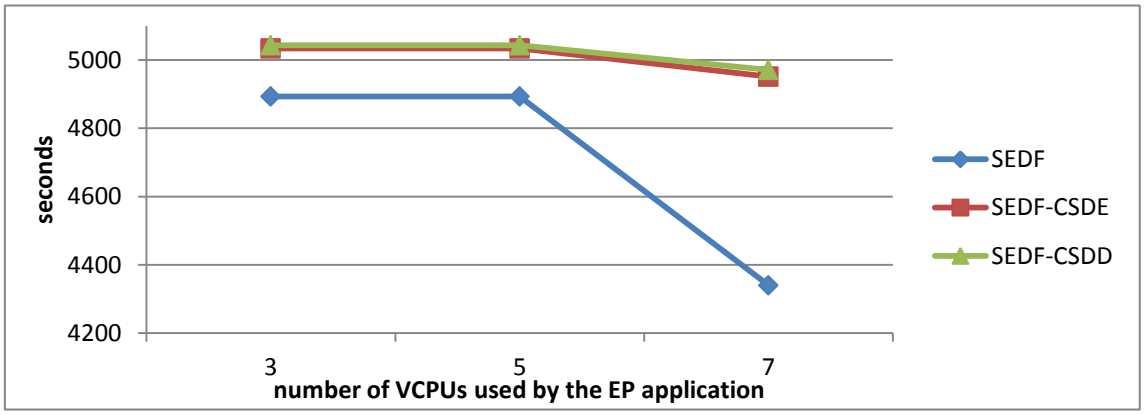


Figure 36 - Execution times for the EP application with various schedulers and various numbers of VCPUs

The lines in Figure 36 - which represent the execution times of EP with the co-scheduling scheduler - are running side by side. The overhead is significantly increasing as the amount of VCPUs is increasing.

6 CONCLUSION

This chapter presents the conclusion of the thesis. The work will be analysed to assess whether it achieved the initial purpose. Further work will be suggested, to reach better results.

6.1 Summary of the performed work

The main goal was to create a co-scheduler solution, which can speed-up the execution of the communicating VMs. The initial purpose was to investigate the current co-scheduling algorithms, which are appropriate to implement in a paravirtualized environment. The architecture of the chosen hypervisor has been explored, and the periodic boost co-scheduling algorithm has been chosen for implementation. The work of Govindan et al. [15, 16] has made a deep impression to the author of the thesis. The author tried to implement a bookkeeping mechanism to gain statistics from the communication behaviour of the VMs. Unfortunately this type of technique became deprecated in the actual version of Xen. After further investigation of the system, it was found that there was a built-in feature to provide statistics regarding the state of the event channels. The solution has been implemented and a cluster environment has been set-up to execute the performance schedulers. Two benchmark programs have been selected for this measurement. The performance test confirmed that the implemented solutions are able to co-schedule the VMs.

6.2 Analysis of the co-scheduler

6.2.1 Performance analyse

As it was stated in the Chapter 5.3.1, the implemented schedulers have $O(N * M)$ time complexity, where N is the number of runnable VCPUs in the RunQ, while M is the length of the array of the event channel. The performance test showed that this overhead appears when many VCPUs share the same CPU.

Furthermore the benchmark test showed that the scheduler successfully co-scheduled the network I/O intensive applications. The schedulers prefer the communication intensive VMs, which increases the possibility to run VMs using a co-scheduled scheme.

6.2.2 Fairness guarantees

If the schedulers prefer the network intensive VMs, then can it happen that two or more VCPUs completely monopolise the CPU by sending messages to each other continuously?

The scheduler is sitting on top of the SEDF scheduler. The SEDF maintains a running queue to store the runnable VCPUs which should run in the given time period. The modified schedulers just reorder this queue by selecting the most network intensive VCPUs in every scheduling cycle. It can have the side effect that the communication intensive VCPUs run at the beginning of each period, while the CPU consuming VCPUs runs at the end. The co-scheduler algorithms guarantee the specified slice for each VCPU. Due to the fact that the schedulers reorder the RunQ, the deadlines of the SEDF schedulers are no longer guaranteed.

6.3 Further works

6.3.1 Extended performance test

The performance test was planned to run on 4 physical servers and up to 24 VMs on each physical computer. We were not able to execute the original test plan because we ran up against IP number and time limitations. The schedulers should be tested with more VCPUs and CPUs. Only two benchmark tests (LU, EP) were used for the measurement because of memory barriers. The test cases should contain higher communicating benchmark programs, like MG or IS.

6.3.2 Decrease the overhead of the schedulers

As mentioned earlier, the schedulers suffer from overheads when a large number of VCPUs are running on a single CPU. Maybe there is a faster way to

figure out the number of pending network messages. An intensive investigation should look for a lower order time complexity algorithm. An alternative solution is that every search stores the second highest intensity VCPUs as well. In this way every second search can skip. After every VCPU search the next chosen VCPU should be the previously found second highest network intensity VCPU. This cuts the overheads in half, but the co-scheduling choice won't be so appropriate.

6.3.3 Implementing own network intensity statistics

The first implementation attempt has been cancelled, because it uses a deprecated technology, namely extend the `shared_info struct.` There should be other shared memory between domains, which can be used for these bookkeeping statistics. The supervisor of this thesis offered to use XenStore, which is an information storage space shared between domains. Jan Beulich from the Xen developer mailing list [34] recommended having domains actively registered in a separate shared data region within the hypervisor, similar to the extension of being able to register the VCPU info area.

Both possibilities require more investigation. Nevertheless if they are working successfully, it allows the implementation of the entire co-scheduling algorithm defined in Chapter 4.3.

REFERENCES

[1] Amazon Web Services (AWS). http://aws.amazon.com [Online; accessed 02-August-2012]

[2] Andrew S. Tanenbaum (2007), “Scheduling”, in: Modern Operating Systems (3rd ed), Prentice Hall, New Jersey, p. 145-163

[3] Anglano, C. (2000), "Comparative evaluation of implicit coscheduling strategies for networks of workstations", *IEEE International Symposium on High Performance Distributed Computing, Proceedings*, pp. 221.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003), "Xen and the art of virtualization", *Operating Systems Review (ACM),* Vol. 37, pp. 164.

[5] Cherkasova, L., and Gardner, R. (2005), “Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor", *ATEC '05, Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 24.

[6] Cherkasova, L., Gupta, D. and Vahdat, A. (2007), "Comparison of the three CPU schedulers in Xen", *Sigmetrics Performance Evaluation Review*, vol. 25, no. 2, pp. 42-51.

[7] Chris Takemura, Luke S. Crawford (2009), "The Book of Xen: A Practical Guide for the System Administrator", No Starch Press, San Francisco

[8] Credit2 Scheduler Development.

http://wiki.xensource.com/xenwiki/Credit2_Scheduler_Development
[Online; accessed 02-August-2012]

[9] David Chisnall (2007), "The Definitive Guide to the Xen Hypervisor", Prentice Hall, Massachusetts

[10] David E. Williams and Juan Garcia (2007), "Virtualization with Xen: Including XenEnterprise, XenServer, and XenExpress", Syngress, Burlington

[11] Bailey, D.H., Barszcz, E., Dagum, L. and Simon, H. D., (1993) "NAS Parallel Benchmark results," *IEEE Parallel and Distributed Technology*, pg. 43-51

[12] David H. Bailey, et. al, (1991) "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, no. 3, pg. 66-73.

[13] Frachtenberg, E., Feitelson, D.,G., Petrini, F. and Fernández, J. (2003), "Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources", pp. 85.

[14] George W. Dunlap, (2011) "Scheduler development update", Citrix Systems R&D Ltd, UK

[15] Govindan, S., Choi, J., Nath, A. R., Das, A., Urgaonkar, B. and Sivasubramaniam, A. (2009), "Xen and Co.: Communication-aware cpu management in consolidated xen-based hosting platforms", *IEEE Transactions on Computers*, vol. 58, no. 8, pp. 1111-1125.

[16] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B. and Sivasubramaniam, A. (2007), "Xen and co.: Communication-aware CPU scheduling for consolidated xen-based hosting platforms", *VEE'07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pp. 126.

[17] History of Xen - The origin of the name http://blog.xen.org/index.php/2008/03/21/history-of-xen-architecture-part-4/ [Online; accessed 02-August-2012]

[18] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N. and Yajnik, S. (2010), "XenTune: Detecting Xen scheduling bottlenecks for media applications", *GLOBECOM - IEEE Global Telecommunications Conference*

[19] Mark Stillwell, (2011), "Dynamic Fractional Resource Scheduling Practical Issues and Future Directions", presentation, *International Research Workshop on Advanced High Performance Computing Systems*, Cetraro, Italy

[20] NAS Parallel Benchmark. http://www.nas.nasa.gov/publications/npb.html [Online; accessed 02-August-2012]

[21] Ousterhout, J. K. (1982), "SCHEDULING TECHNIQUES FOR CONCURRENT SYSTEMS.", *Proceedings - International Conference on Distributed Computing Systems,* pp. 22.

[22] Petrini, F. and Feng, W. (2000), "Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems", Proceedings of the International Parallel Processing Symposium, IPPS, pp. 439-444.

[23] Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y. and Pu, C. (2010), "Understanding performance interference of I/O workload in virtualized cloud environments", *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*, pp. 51.

[24] RT-XEN: Real-Time Virtualization Based on Hierarchical Scheduling. https://sites.google.com/site/realtimexen/ [Online; accessed 02-August-2012]

[25] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam and Chita R. Das, (1999) "A closer look at coscheduling approaches for a network of workstations", *In Eleventh ACM Symposium on Parallel Algorithms and Architectures*, SPAA'99

[26] Sodan, A. C. and Lan, L. (2005), "LOMARC - Lookahead matchmaking for multi-resource coscheduling", *Lecture Notes in Computer Science*, Vol. 3277, pp. 288.

[27] Sodan, A.,C. and Huang, X. (2006), "Adaptive time/space sharing with SCOJO", *International Journal of High Performance Computing and Networking*, vol. 4, no. 5/6, pp. 256-269.

[28] Steven H. VanderLeest, (2012), "ARINC 653 HYPERVISOR", DornerWorks, Ltd.

[29] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/ [Online; accessed 02-August-2012]

[30] The Xen Hypervisor. http://xen.org [Online; accessed 02-August-2012]

[31] White, S., Ålund, A., and Sunderam, V. S., (1995) "Performance of the NAS parallel benchmarks on PVM-based networks." *Journal of Parallel and Distributed Computing.* 26, 1 (Apr. 1995), pp. 61-71.

[32] William von Hagen (2008), Professional Xen Virtualization, John Wiley & Sons, Indianapolis

[33] Xen and Co.: Communication-Aware CPU Scheduling and Accounting in Xen: http://csl.cse.psu.edu/?q=node/54

[Online; accessed 12-August-2012]

[34] Xen developer discussion. http://lists.xen.org/mailman/listinfo/xen-devel [Online; accessed 12-August-2012]

[35] Xi, S., Wilson, J., Lu, C. and Gill, C. (2011), "RT-Xen: Towards real-time hypervisor scheduling in Xen", *Embedded Systems Week 2011, ESWEEK 2011 - Proceedings of the 9th ACM International Conference on Embedded Software*, EMSOFT'11, pp. 39.

[36] Xianghua Xu, Peipei Shan, Jian Wan, and Yucheng Jiang. (2008) "Performance Evaluation of the CPU Scheduler in XEN." *Proceedings of the 2008 International Symposium on Information Science and Engineering - Volume 02* (ISISE '08), IEEE Computer Society, 68-72.

APPENDICES

Appendix A Appendix Source Code

A.1 Specify the number of pending messages

```
/* Co-Sched changes starts ------------------------------------- */
static int get_number_of_messages(const struct domain *d)
{
   int number_of_messages=0;
   for (int port = 0; port < MAX_EVTCHNS(d); port++)
   if (d->shared_info->evtchn_pending[port] == EVTCHNSTAT_virq)
      number_of_messages++;
   return number_of_messages;
}
/* Co-Sched changes ends  ------------------------------------- */
```

A.2 Scheduler modification

```
static struct task_slice sedf_do_schedule( const struct scheduler *ops,
      s_time_t now, bool_t tasklet_work_scheduled)
{

   ...

   /* Co-Sched changes starts ------------------------------------- */
   int run_num_of_messages = 0,
      temp_num_of_messages,
      element_id, runinf_id;
   struct list_head *elementq;
   struct sedf_vcpu_info *element_inf = NULL;
   /* Co-Sched changes ends --------------------------------------- */

   ...

   runinf = list_entry(runq->next,struct sedf_vcpu_info,list);

   /* Co-Sched changes starts ------------------------------------- */
   runinf_id = runinf->vcpu->domain->domain_id;
   run_num_of_messages = get_number_of_messages(runinf->vcpu->domain);
   if (!list_empty(runq))
   {
      list_for_each(elementq,runq)
      {
         element_inf = list_entry(elementq, struct sedf_vcpu_info, list);
         element_id  = element_inf->vcpu->domain->domain_id;
         if (runinf != element_inf)
         {
            temp_num_of_messages =
            get_number_of_messages(element_inf->vcpu->domain);
            if (element_id == 0)
            {
               /* more sensitive to domain0 */
               if (temp_num_of_messages > 0)
               {
```

```
                    runinf    = element_inf;
                    runinf_id = element_id;
                    break;
                }
            }
            else if (runinf_id == 0)
            {
                /* more sensitive to domain0 */
                if ((run_num_of_messages == 0) && (temp_num_of_messages >0))
                {
                    runinf    = element_inf;
                    runinf_id = element_id;
                    run_num_of_messages = temp_num_of_messages;
                }
            }
            else
            {
                if ((run_num_of_messages < temp_num_of_messages) ||
                    /*start that one, which wait for longer time for fairness*/
                    (run_num_of_messages == temp_num_of_messages) &&
                    (runinf->sched_start_abs > element_inf->sched_start_abs)))
                {
                    runinf    = element_inf;
                    runinf_id = element_id;
                    run_num_of_messages = temp_num_of_messages;
                }
            }
        }
    }
}
/* Co-Sched changes ends ---------------------------------------- */

...
ret.task = runinf->vcpu;
return ret;
}
```