August 2013
Master’s Degree Thesis

A Space-Shared Scheduler in Microkernel for Many-Core Systems

Graduate School of Chosun University

Department of Computer Engineering

Ganis Zulfa Santoso

A Space-Shared Scheduler in Microkernel for Many-Core Systems

다중 코어 시스템을 위한 마이크로 커널의 공간
공유 스케줄러 구현 및 성능평가

August 23, 2013

Graduate School of Chosun University

Department of Computer Engineering

Ganis Zulfa Santoso

A Space-Shared Scheduler in Microkernel for Many-Core Systems

Advisor: Prof. Moonsoo Kang, PhD

A thesis submitted in partial fulfillment of the requirements for a Master's degree

April 2013

Graduate School of Chosun University

Department of Computer Engineering

Ganis Zulfa Santoso

산토소 가니스 술파의
석사학위논문을 인준함

위원장	조선대학교 교수	모상만	(인)
위　원	조선대학교 교수	심재홍	(인)
위　원	조선대학교 교수	[illegible]	(인)

2013 년 5 월

조선대학교 대학원

Table of Contents

List of Tables

List of Figures

Acronyms

CPU	Core Processing Unit
GB	Gigabytes
ISA	Instruction Set Architecture
LCPU	Logical CPU
L1	Level 1
L2	Level 2
MB	Megabytes
OS	Operating System
PCPU	Physical CPU
RE	Runtime Environment
VM	Virtual Machines

한 글 요 약

다중 코어 시스템을 위한 마이크로 커널의 공간 공유 스케줄러 구현 및 성능평가

산토소 가니스 술파
지도 교수: 강문수
컴퓨터공학과
대학원, 조선대학교

차세대 임베디드 시스템으로 다중 코어 시스템이 산업계의 주목을 받고 있다. 베이스칩 상의 코어의 개수 증가는 개별 코어의 메모리 접근 대역폭의 감소와 코어 당 실행스레드 수 감소와 같은 문제들을 새롭게 야기했다. 이러한 문제를 해결하기 위해 공간공유 일정 스케쥴링과 같은 패러다임이 프로세서 스케줄링의 한가지 방법으로 제안되었다. 공간 공유 스케줄링의 기본적인 아이디어는 응용 프로그램에 인접한 복수의 코어들을 할당하고 응용 프로그램이 자체 자원을 관리하도록 하는 것이다.

현재 대부분의 운영 체제는 모놀리틱 커널이 일반적이며, 공간 공유 스케쥴링 대신, 시간 공유 스케쥴링을 기반으로 하고 있다. 이러한 형태의 운영 체제는 다중 코어에 적합하지 않으며, 코어의 수에 따른 확장성을 가지고 있지 않다. 이러한 이유는 속도의 성능과 안정성에서 최근 주목할만한 발전을 보인 마이크로 커널이 다중 코어를 활용하기 위한 운영체제로 주목을 받고 있다.

본 학위 논문에서는 다중 코어 시스템을 타겟으로 하는 마이크로 커널 상에서 공간 공유 스케쥴러를, L4/Fiasco 내부에 구현하고 그 성능은 CPU 계산 집약적인 병렬 태스크들을 사용하여 평가하였다. 평가 결과는 L4/Fiasco의 시간 공유 스케쥴링과 공간 공유 스케쥴링의 조합이 다중 코어 시스템에 적합하며 코어 수에 따른 확장성도 뛰어나다는 것을 보여준다.

I. Introduction

Many-core system is something that is unavoidable due to the limit on CPU clock frequency. To counter the problem, the industry and academic communities agreed to increase the number of cores in a chip to increase performance. The increasing number of cores on a single chip introduced new issues such as decrease in memory bandwidth and less number of threads per core. A new paradigm should be implemented on the processor scheduling.

Current Operating Systems are only employing time-sharing scheduling. Time-sharing scheduling solely are not scalable for many-core systems. It is developed based on the idea to optimize processor resources, in many-core however, memory access resources are more expensive.

A. Research Objective

Space-shared scheduling has been proposed. The basic idea of such scheduling is to allocate a number of cores to an application and let the application manages its own resources. It optimize the memory usage and latency and reducing the OS complexity instead of traditional usage efficiency.

We implement a space shared scheduler inside a microkernel, L4/Fiasco. Microkernel is chosen because its capability in creating a resilient and robust Operating System. Space-shared scheduler will aid time-shared scheduler in organizing tasks. Its performance will be evaluated using CPU-intensive tasks. The scheduler is extended into and emulated in a Linux environment to observe how it behaves in a popular Operating System. The results show complementing space-sharing scheduling and time-sharing scheduling in L4/Fiasco is scalable for many-core environment.

B. Thesis Layout

We first study the key concepts in scheduling for many-core environments in literature. In chapter II, we first discuss the trends that realized many-core concepts. The related works are discussed in chapter III. In chapter IV, we explain and discuss our proposed space-shared scheduler. The performance evaluation is presented in chapter V.

II. Background Concept

Moore’s Law predicts the number of transistors doubling approximately every two years. This law is still held right now, but the clock frequency remains fairly constant since 2010 shown by Figure 2-1.

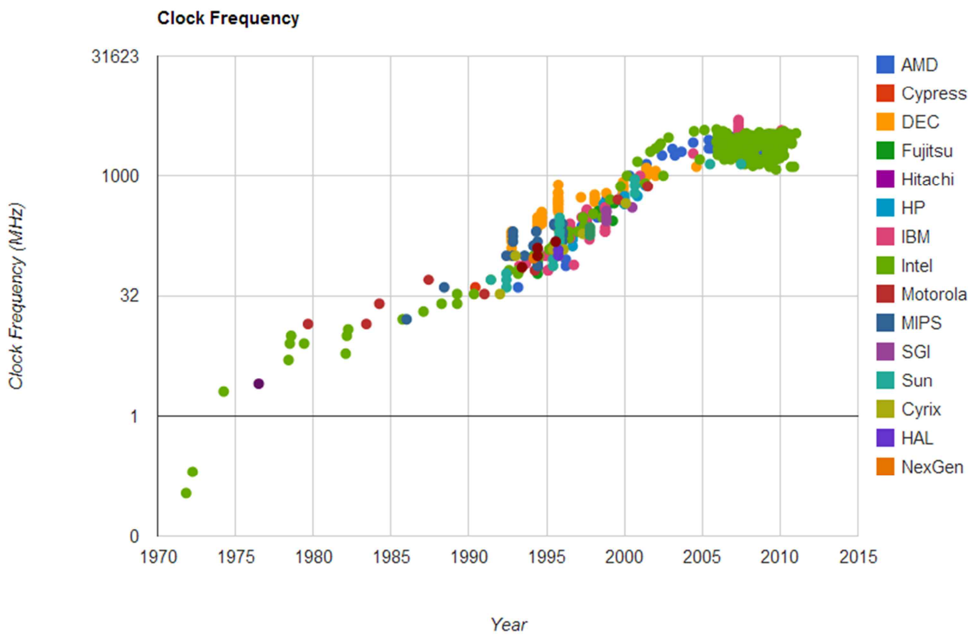

Figure 2-1: The clock frequency remains constant.

A. Many-Core system

Even though we can still build transistor smaller and faster, we can’t increase the clock frequency because it will generate a lot heat to turn on billions of transistors in the same time. Instead of creating faster and complex processors, we are now creating more and simpler processors. That is why the emergence of multi-core systems cannot be evaded.

Many-core system is a multi-core system but it is loosely defined as having more than 50 cores on single chip. At around 2004, the maximum clock frequency of a single processor is around 4 GHz. In the term of economy and engineering, it is not suitable to push the frequency of a single core even more. Rather than having one powerful single core running at the high frequency, it is more efficient to have multiple cores running at lower frequencies. The academic and industry agreed that to conform to the Moore’s Law, the number of processors in a single chip should be increased. The trend of one single core towards multiple cores was begun at 2005 with the introduction of dual cores, four cores at 2008 and the last one was quad cores at 2010. The most obvious method of moving forward is to start increasing the more cores to the future of chip designs. The trend of multi-core processors can be seen in the Figure 2-2. Responding to the new trend of many-core processors, Intel introduced Xeon Phi with number of cores is more than 60 cores.

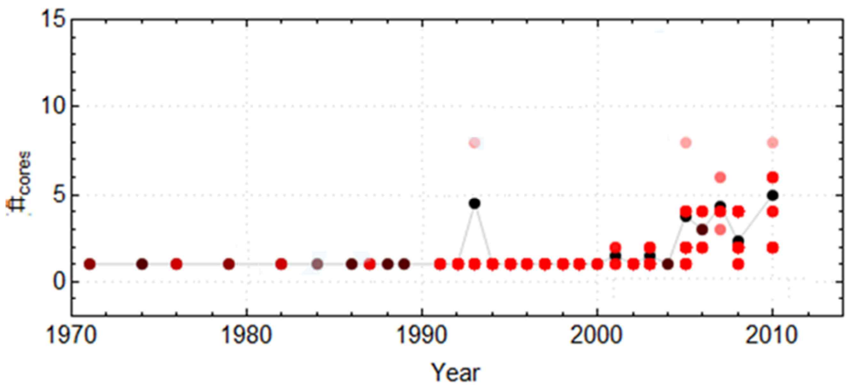

Figure 2-2: Increasing number of cores on processors.

Many-core system, despite the similarity with multi-core system, introduced new challenges. The current paradigm that we have on Operating Systems realized because of the characteristic of single-core; multiplex the CPU. However, in many-core system we will have much more than one core and possibly multiplexing CPU is not necessary. Therefore the advent of new paradigm is needed.

B. Space-Shared Scheduler

In the introduction of many-core chips, there are three trends that can be seen. The first trend is instead of fewer and bigger cores, the shifting is moving towards simpler and more cores. Another one is heterogeneous architecture. It means each core will have specialized capabilities and packed onto the same chip. The last predicted trend is with the rise of memory technology, such as embedded RAM, so there will be more memory to be placed on the chips.

These trends will introduce new problems that never had been encountered before. But we will focus on the first trend as it is relevant with our sub section topic. The first trend opens up the possibilities that time-sharing is no longer needed as each process can be run in its own core, or put it another way is there will be more cores than the number of threads. Time-sharing will become a bottleneck and unnecessary complexity in the operating system kernel.

The idea of space-sharing computer resources is emerged to answer this challenge. In principle, it seeks optimization in term of memory usage and reduction of OS complexity. This is a new paradigm as we used to try to make core usage as efficient as possible.

Space shared scheduler allocates a number of cores to one application or virtual machine and let it manages it as it is its own resources. The scheduler will not interfere or try to micromanage it. The concept is new and no popular OS is already adapted to this concept. Even more, they are not suitable for this paradigm since they are built around monolithic kernel architecture.

C. Microkernel

Kernel is the core component of an operating system. It resides between applications and actual data processing at the hardware level. Early computer systems did not have kernel but if we are going to run multiple programs on a single computer, kernel is substantial. The most famous used of kernel architecture is monolithic kernel which is used by Linux, BSD, and MS-DOS.

At the beginning, the size of kernel was generally small. But as the number of features, and broad range of devices were added to the kernel, the size began to grow. As for Linux kernel 3.8[9], currently there are more than 16 million total of lines of code, which dominated by drivers, file system and architecture-specific code. In that scale, it is almost impossible to avoid bugs and security leaks. It is even worse since most of them are running in the kernel mode. Bug-prone code in the kernel mode can corrupt the operations of the whole system.

Microkernel is architecture of kernel which provides minimum functions run in the kernel mode. The counterpart of microkernel is monolithic kernel. Unlike microkernel, most of the code of monolithic kernel is run in the kernel mode. The differences of both of them can be seen clearly by the Figure 2-3.

Microkernel provides mechanisms to execute programs and enforcing isolations between them but it does not include complex services. The other components operate in the user mode. Therefore the size of kernel is small. As an example, L4/Fiasco has only 20,000 lines of code. The kernel isolates the device drivers using virtual address spaces to minimize the bug-prone code's effects on the whole system.

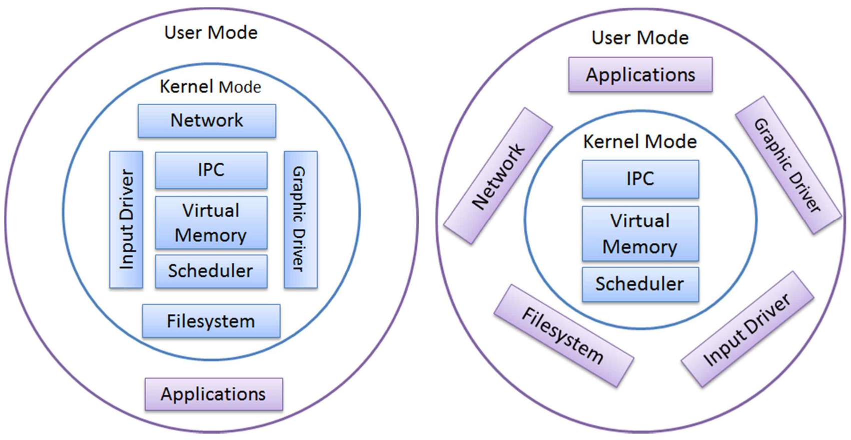

Figure 2-3: Architecture of monolithic kernel vs. microkernel.

The advantages of microkernel are it is more robust, more reliable and it can provide real time operating systems for users. Therefore most of the applications of microkernel are for reliability-sensitive markets such as military, automotive, and aerospace industry.

The monolithic kernel architecture has an obvious advantage over microkernel which is its superior performance. Since the entire kernel components are run in the kernel mode, the context switch from user mode to kernel mode and vice versa are less compared to microkernel. This characteristic was demonstrated by the poor performance of the earliest examples of the microkernel such as Mach and Chorus OS. The latest generation of microkernel however successfully reduces the performance gap between these architectures.

Unlike monolithic kernel, microkernel can isolate the applications. If one of the applications is corrupted, it will not destroy the whole systems. This characteristic makes microkernel is very suitable to create an OS with a space-shared scheduler.

D. L4/Fiasco

L4 Microkernel is one of the prominent microkernels in the research and academic community. It was created by Jochen Liedtke to address the poor performance of the earlier microkernels. He fixed the inaccurate concepts of Mach and simplified the concepts. L4's concept of microkernel is a microkernel does no real work as it only provides inevitable mechanisms and no policies implemented.

L4 Microkernel is further researched by various universities and companies. A commercial example of L4 variant is OKL4 from OK Labs which its product has been deployed in more than 1.5 billion devices [10]. Another notable example is L4Ka::Pistachio which is developed together by University of Karlsruhe, Germany and University of New South Wales, Australia.

L4/Fiasco or Fiasco.OC which is used in this thesis is developed by Hermann Hartig's group at the Technical University of Dresden, Germany. His group also developed operating systems such as DROPS [12] and NIZZA [11] on top of L4 beside L4Linux which will be discussed on next section. L4/Fiasco can be run a broad range of hardware including in ARM systems such as Freescale i.MX6 board which will be used in this thesis. L4/Fiasco is an open source project and can be easily downloaded from the homepage [13].

The general architecture of Fiasco is demonstrated on Figure 2-4. Basically there are three components in the system of L4/Fiasco: Microkernel, Runtime Environment and Applications. Microkernel is the only component that is run in the kernel mode whilst others are run in the user mode. The general design of microkernel is to make it as small as possible to make it more robust against bugs and attacks. It only consists of simple services and does not include services for accessing hardware, bootstrapping, etc. Therefore L4Re provides servers to help the operations of Fiasco microkernel and to build applications on top of microkernel

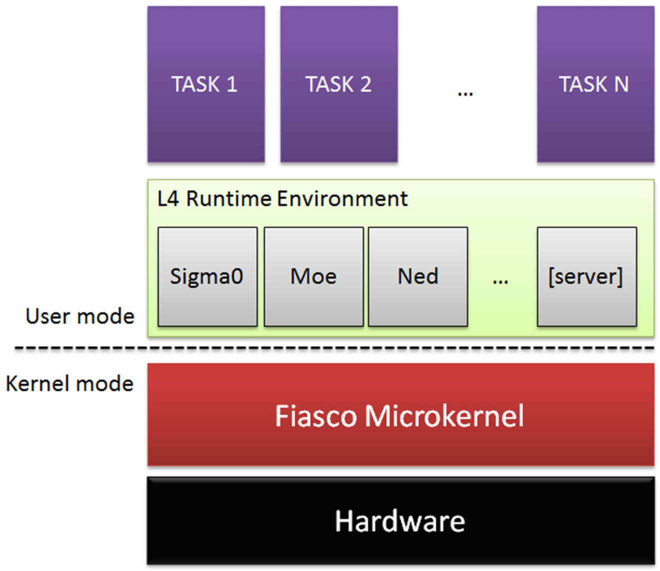

Figure 2-4: General architecture of L4/Fiasco.

L4Re consists of several servers: Sigma0, Moe, Ned, Io, Mag, fb-drv and Rtc. The first two is needed to be booted beforehand. Sigma0's task is to resolve page fault. Moe's tasks are to bootstrap the system and to provide resources managements. For further on, the Fiasco.OC and L4Re will be discussed as the same entity.

E. L4Linux

L4Linux were ported to Linux kernel by treating it like another port of Linux to a new architecture. Currently, L4Linux has been updated to the Linux Kernel 3.8. The design overview for the L4Linux is depicted in the following Figure 2-5. L4Linux can be run side by side with other real-time operating systems on top of

L4/Fiasco. The real time operating system's proper operation will not be violated as L4Linux is run on user mode and cannot override the operation of L4/Fiasco.

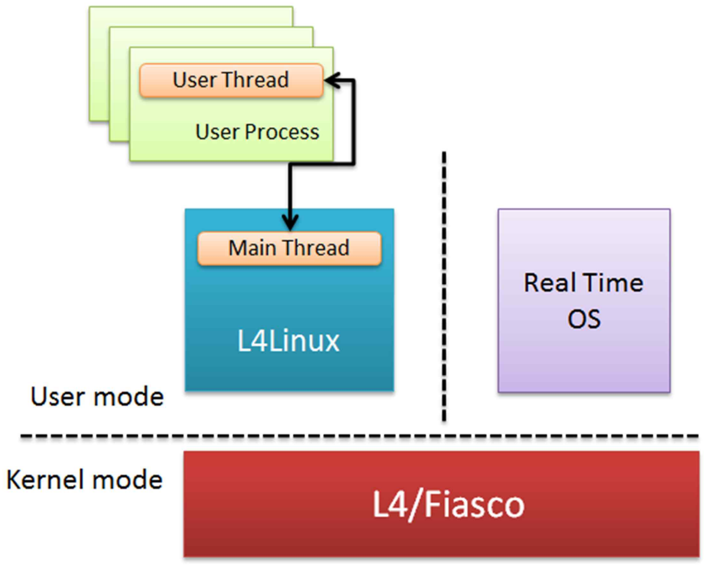

Figure 2-5: General design of L4Linux.

By having L4Linux ported to L4/Fiaso, developers are spoiled with the abundant legacy applications of Linux. Real-time OS can be used to video or audio applications. By running it in different OS and on top of L4/Fiasco, it is assured that the application operations will not be interrupted.

F. ARM architecture

In the world of mobile phone and embedded devices, ARM architecture dominates by having 95% of the industry. ARM was designed for mobile and low power computing segment with its power efficient processors. But now with the introduction of Cortex A9 and A15 designs, it explores a new territory of high end desktop and server markets. The ISA (Instruction Set Architecture) of ARM is 32 bit RISC architecture. It is a fixed instruction width and dominated by single cycle execution. Other vendors are trying to modify their current processor to become more power efficient, the ARM on the other hand is trying to upgrade the performance of their current processor from already power efficient processors. In the term of many-core processors, it seems that ARM architecture is gaining tractions.

G. Freescale i.MX6

i.MX series are based on ARM Cortex A9 architecture and offering from single-core up until quad-core. The one that will be utilized in this thesis is the quad-core version and therefore the main discussion will be around this version. The version will running up to 1.2 GHz with 1 MB of L2 cache, and 64-bit DDR or 2-channel, 32-bit LPDDR2 support. It supports up to 1080p video playback and it has 3D engine to perform computational tasks.

III. Related Works

The trend of many-core processors has piqued interests from academic and research community. Therefore experimental operating systems have been researched extensively in universities. But as far as the writers' knowledge, there is no research written to create a space scheduler mechanism on top of a microkernel, especially L4/Fiasco. However, the creation of space shared scheduler on top of another type of kernel is not a novelty.

Tessellation

From Berkeley University, there is Tessellation [3] which is developed at Parallel Computing Laboratory. It is based on two fundamental ideas: Space-Time Partitioning (STP) and Two-Level Scheduling. Some functionality of Tessellation is similar to the hypervisor as it combines the space-sharing and time-sharing concept. In Tesselation, the space-sharing divides cpus resources into *partitions* and it can be fixed or dynamically sized. An application will be given a privilege to modify the number of its own resources. One of the differences with our thesis is Tessellation is using exokernel instead of microkernel.

Corey OS

Corey [6] is another exokernel-based Operating System and created by Parallel & Distributed Operating Systems Group of MIT. The first integral idea in Corey is cores use only local resources and share no state between them. Utilizing the interface of the kernel, applications should create sharing whenever necessary. Another fundamental concept is physical memory in Corey is described as *address trees*. Address space of each core will be identified by an address tree. It implies that there is no need for a global memory structure in OS. The last key method of

Corey is its core management and scheduling. It implements a space-share scheduling mechanism.

fOS

From MIT there is another operating system called as fOS [8]. It is based on its three design concepts: (i) share nothing, communicate only by messaging, (ii) space-sharing instead of time-sharing, (iii) fault tolerance by automatically re-routing tasks in the same group of cores.

Barrelfish

From the land of Europe, ETH in Zurich and Microsoft research collaborated to create Barrelfish [7]. It utilized another type of kernel called as multikernel. The three design principles of Barrelfish are: (i) making all inter-core communication explicit, (ii) making the OS structure HW neutral, and (iii) viewing state as replicated instead of shared. The main disadvantage of Barrelfish is its memory management sub-system. It puts unnecessary complexity into the user-space applications without actually gaining substantial performance improvements.

Singularity

Based on its own previous research OS called Singularity, Microsoft research developed HeliOS [14]. It is using a so-called satellite kernel. It is embracing heterogeneity but also using space-sharing mechanism to reduce the issues in many-core environment. The system consists of three fundamental features: (i) software-isolated processes, (ii) contract-based channels, and (iii) manifest-based programs.

IV. Proposed Space-Shared Scheduler

The goal of the scheduler is to provide a space shared scheduler mechanism in microkernel to overcome the challenges in many-core systems. The scheduler will be implemented in the L4/Fiasco, a microkernel developed in TU Dresden. It will communicate with applications on top of microkernel and provides sets of CPU that can be used by each application.

A. Design

Current scheduler of L4/Fiasco is using time-shared scheduling with round-robin mechanisms for tasks with the same priority. The implemented design will not affect the time shared-scheduling, it will add the space shared scheduling to microkernel. The scheduler is designed as small and as concise as possible. It employs minimum modifications to the current system of L4/Fiasco.

The scheduler is implemented in the microkernel. For every application on top of microkernel, it should communicate with the scheduler to get the physical id of CPUs (PCPU) that can be used. Further on, the PCPU values that are given by the scheduler is mapped to the logical CPU in each application. The scheduler inside the application will manage its own resources by itself. Microkernel will not override the mechanism inside the applications. Microkernel will only employ the time-shared scheduling if there is more than one thread running in the same physical core. The space shared scheduler is called at the booting of the applications. The figure of Figure 4-1 provides explanation on how the mechanism works in the initialization phase of the applications. After getting the IDs of PCPU, an application does not have to communicate with the space-share scheduler every time it runs a thread.

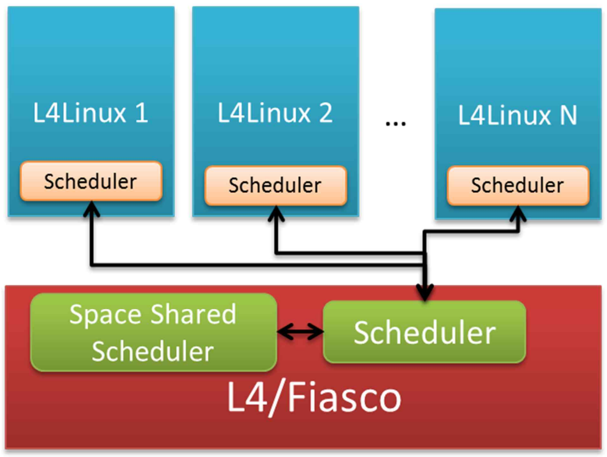

Figure 4-1: The proposed space-shared scheduler mechanism in L4/Fiasco.

Further on, an application can communicate with the scheduler if it needs to change its own CPU resources. The scheduler will give the new value of PCPU to the respective application and also other applications if there are changes as well.

The processes within an application may communicate with each other therefore space-shared scheduler takes into account the affinity of the CPUs. The scheduler will try to give the most optimum of CPU allocation to the applications by giving a set of CPUs to a single application with best affinity. Once L4Linux has mapped the PCPU and LCPU, a thread will be spawned for each of the connection. This mechanism can be seen in Figure 4-2.

L4Linux will manage those threads just like its own CPUs. The scheduler on L4Linux will manage on how the applications on top of L4Linux run. Scheduler on L4/Fiasco will not interfere on the scheduler in L4Linux. The scheduler in L4/Fiasco will multiplex if there is more than one thread accessing a PCPU with the same priority. A real time operating system can be realized in this system by giving it higher priority than non-real-time operating systems.

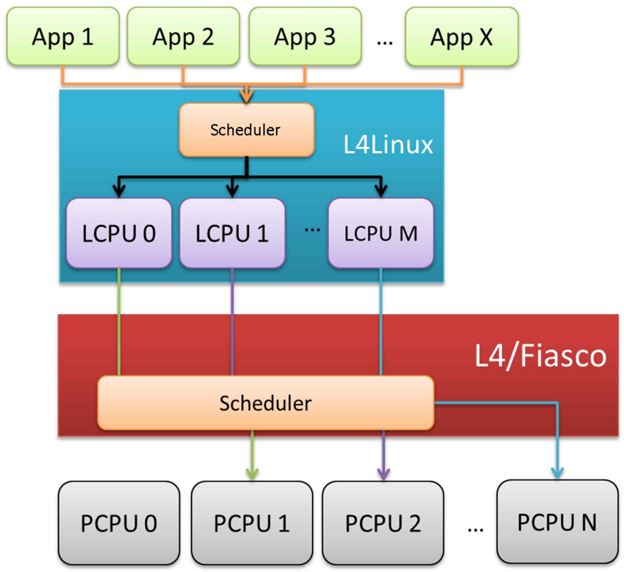

Figure 4-2: An example of mapping mechanism of LCPU and PCPU.

B. Procedures

Once the space-shared scheduler has been implemented, Virtual Machines (VM) on top of fiasco have to contact the space shared scheduler module after the booting. In the initiation, VM has to tell fiasco how much processors it needs. If the modules have not received requests from all VMs, it will not issue CPUs to any VM. If it has received all requests, it will issue the CPUs. The procedure is clearly explained by following Figure 4-3 and Figure 4-4. In this example we are going to use two L4Linuxes.

During the wait, L4Linux 1 has to sleep for few milliseconds, and it asked Fiasco again whether a set of CPUs has been prepared for it. If Fiasco is ready, it will give

L4Linux 1 a set of CPUs, if not L4Linux 1 has to wait for one more time. The procedure is presented in Figure 4-5.

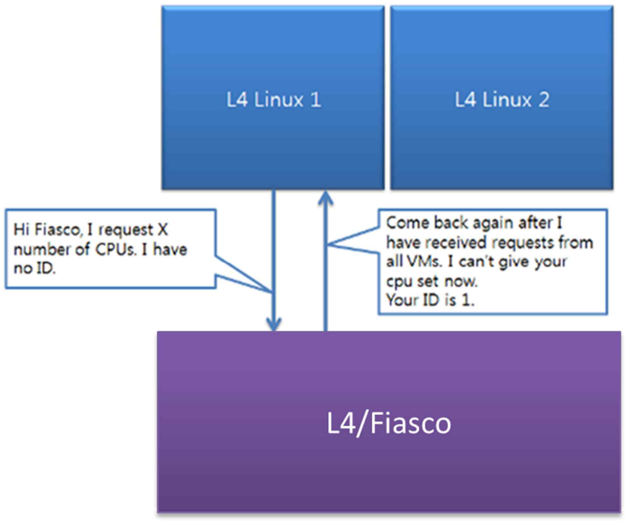
L4 Linux 1
L4 Linux 2
Hi Fiasco, I request X number of CPUs. I have no ID.
Come back again after I have received requests from all VMs. I can't give your cpu set now. Your ID is 1.
L4/Fiasco

Figure 4-3: L4Linux 1 requested a number of CPUs and told to wait by space-shared module and received an ID to request later on.

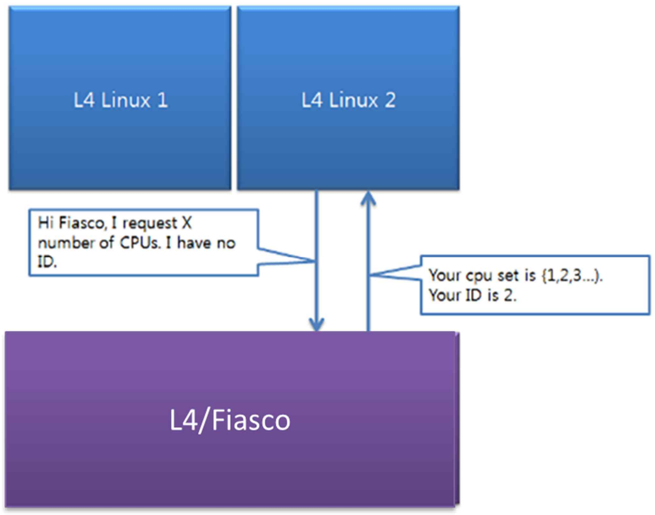
L4 Linux 1
L4 Linux 2
Hi Fiasco, I request X number of CPUs. I have no ID.
Your cpu set is {1,2,3...}. Your ID is 2.
L4/Fiasco

Figure 4-4: L4Linux 2 requested a number of CPUs and received a set of CPUs and its own ID.

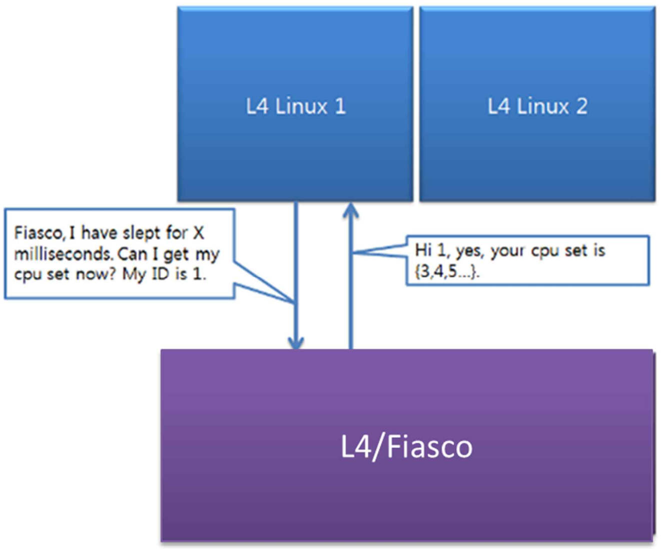
L4 Linux 1
L4 Linux 2
Fiasco, I have slept for X milliseconds. Can I get my cpu set now? My ID is 1.
Hi 1, yes, your cpu set is {3,4,5...}.
L4/Fiasco

Figure 4-5: L4Linux 1 asked Fiasco again and given a set of CPUs to it.

The space shared module is active throughout the time waiting for any further requests from VMs to change its number of CPU sets.

C. Implementation

To efficiently implement the module we need to now the architecture of processors. A characteristic such as CPU affinity is unique for all processor architectures. Having investigated the manual of ARM Cortex A9 [15], we can see the layout and cache hierarchy of this architecture. Each CPU has its own L1 cache. L2 cache is shared among all CPUs. The CPUs are laid serially, not in form of a rectangular. We have to take into account these characteristics while implementing a space shared scheduler.

There are two layers between hardware and Virtual Machines in Fiasco architecture. Thus the implementation should be done in each layer. The modifications are cross-platform and are not dependent on the implemented VMs. However, since it is a paravirtualization, the VMs layer is also need to be modified to adjust to the module. And different VMs have different modifications. In example, the implementation in Linux will be different with the implementation of another open source OSs. In this thesis we will show how the implementation is done in Linux, or more specifically L4Linux.

Fiasco Layer

The scheduler in Fiasco is implemented in this layer, so our modification is largely located in here. The space-shared scheduler is regarded as an additional feature so we are not changing anything on the main class of Scheduler in fiasco. We add a variable and a function which is connected to the space shared scheduler. The details of the space-shared module and the relation with the scheduler object of Fiasco are explained in Figure 4-6.

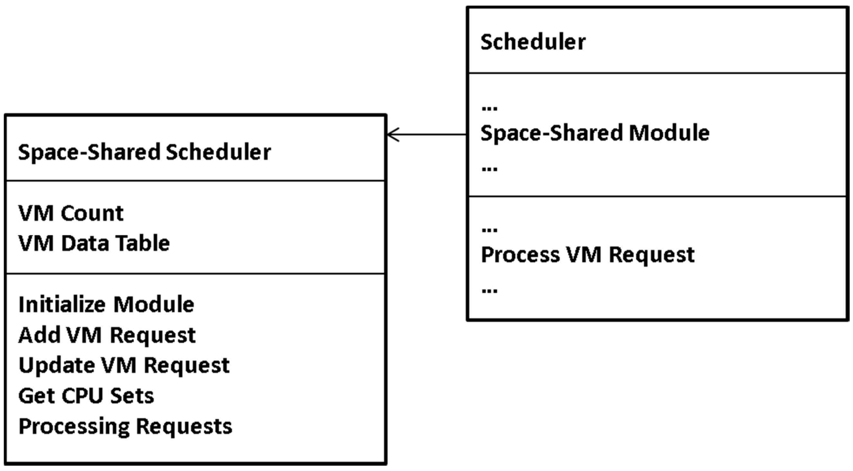

Figure 4-6: The relation of space-shared scheduler object with the native scheduler object.

Space Shared Scheduler Class

VM Count: It tracks how many VMs will connect to the Fiasco and once it is fulfilled it runs the Processing Requests method.

VM Data Table: It contains the data of VMs ID, requested number of CPU, and given CPU sets.

Initialize Module: It sets the initial value of variables in the module,

Add VM Request: A method to get the request of a VM and add it to the VM Data Table.

Update VM Request: It accepts the request of a VM to change its request.

Get CPU Sets: It returns the CPU sets of a VM.

Processing Requests: This method calculates the most optimum CPU Sets for each VM given the data in VM Data Table.

Native Scheduler Class:

Space Shared Module: The interface to the space shared object.

Processing VM Request: It is actually only passes the request of VMs to the space shared module. The method itself does not do anything.

L4Re Layer

In L4Re layer, the scheduler is only an abstraction object. Therefore the only added implementation is a function called l4_scheduler_req_cpu. It is a function that connects the VMs and Kernel and provides no specific operations at all.

L4Linux Layer

The implementation in this layer has only three specific jobs:

(1) Tell space share module how much processors it needs,

(2) Get the physical id of CPU that is given by the space shared module, and

(3) Insert the values to the native mapping of physical and logical CPU.

Specific to L4Linux, these three jobs can be added in the l4x_cpu_virt_phys_map_init() function. We implemented the code that calls l4_scheduler_req_cpu() in L4Re in this function. L4Linux has its own system of mapping the physical and logical CPUs, hence we do not need to create one and only added values that will be used by the mapping. In conclusions, the tasks or L4Linuxes communicate with the scheduler objects trough the abstraction in the L4Re layer and then it invokes the object in the microkernel. The mechanism is shown in Figure 4-7.

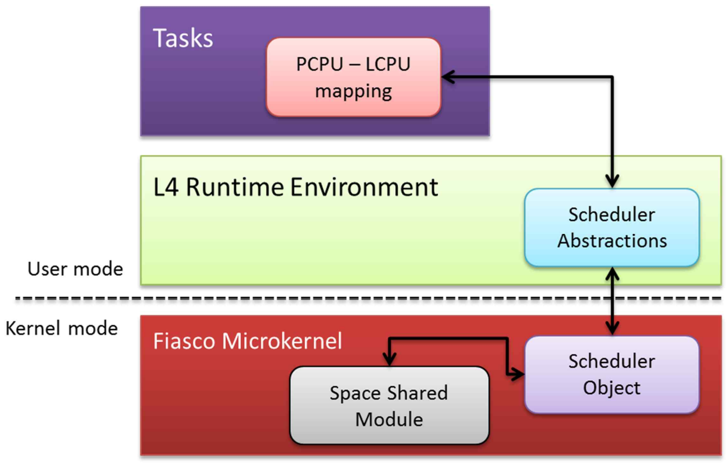

Figure 4-7: The communication between tasks and the scheduler object in microkernel.

The details of modifications are explained line by line in the appendix. How to run along with the configurations are also stated there.

V. Performance Evaluation

A. Many-Core Evaluation in L4/Fiasco

First one we are going to observe the scalability of L4/Fiasco in many-core systems. In here we are going to run a CPU-intensive task on a single core as a baseline as shown in Figure 5-1. It is compared to how fast the tasks are run in a multi-core or many-core architecture as describe in Figure 5-2.

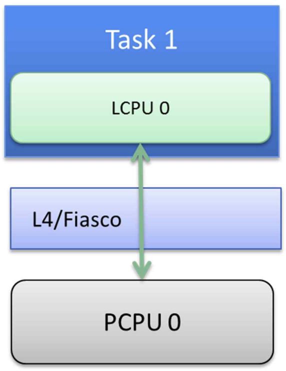

Figure 5-1: Running a task in a single-core.

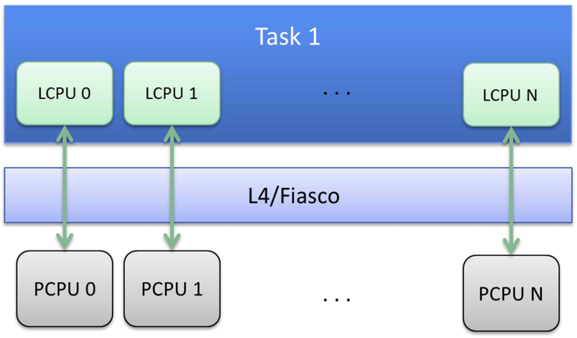

Figure 5-2: Running a task in a multi-core.

The parameters for this simulation are summarized in table 5-1. In this emulation, we are using a popular open source emulation tool called as QEMU.

Table 5-1 Emulation parameters for many-core evaluation in L4/Fiasco

Parameter		**Value**
Emulation Tool		QEMU 0.14
L4Re Version		February 2013 release
L4Linux Kernel Version		Linux 3.7
Host	Machine	Intel Xeon® E5606
	Host CPU Architecture	x86 64 Bit
	Number of Processor	8
	Memory	12 GB
Guest	Architecture	i386
	Number of Cores	8, 16. 32. 64
	Memory	383 MB
	Number of VMs	8, 16. 32. 64

The evaluation results are shown in Figure 5-3. Even though the performance is increasing when we are using more cores, but it does not scale well with the number of cores.

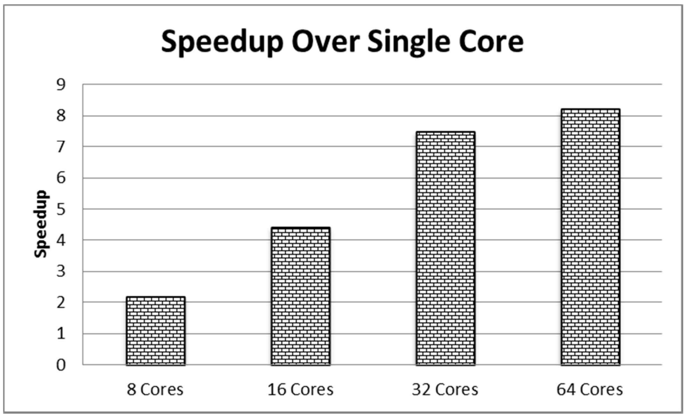

Figure 5-3: Speedup over single-core.

In a multi-core environment the increase of cores is expected to be linear with the performance increase, of course we have to considering the constraint which are posed by the Amdahl's law. The speed up parameter which is used by the following figure is calculated by the dividing the execution time of single core by execution time of *N* core. With *N* is 8, 16, 32 and 64. From 8 to 16 and 16 to 32 we can see that the speed up is almost doubling just like the number of cores. But from 32 cores to 64 cores, the increase is not double. This is an indication that the current scheduling is not scalable for many-core systems.

B. Space-Sharing Scheduling Evaluation

We are going to evaluate our implemented space-sharing scheduler. For this we are going to compare the performance of our Fiasco/L4 with the addition of space-sharing scheduling against the original Fiasco/L4. The original Fiasco/L4 is described in Figure 5-4 while our modification is shown in Figure 5-5. The figures are only example with 4 cores; in the emulation we are going to use 8, 16, 32, and 64 cores.

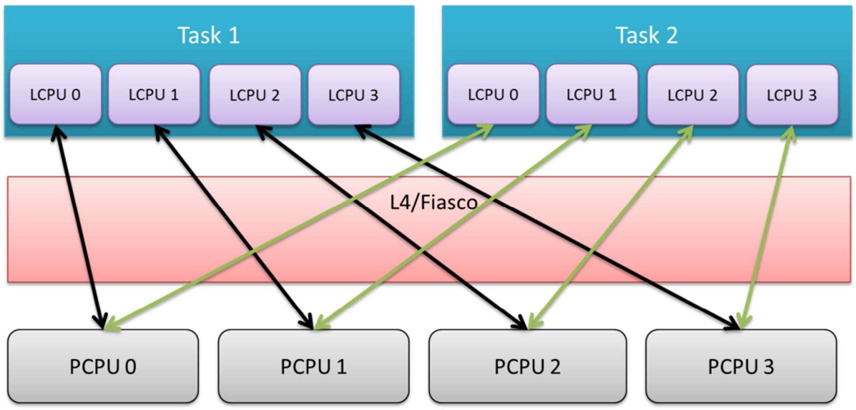

Figure 5-4: A scenario of original L4/Fiasco without space-shared scheduler.

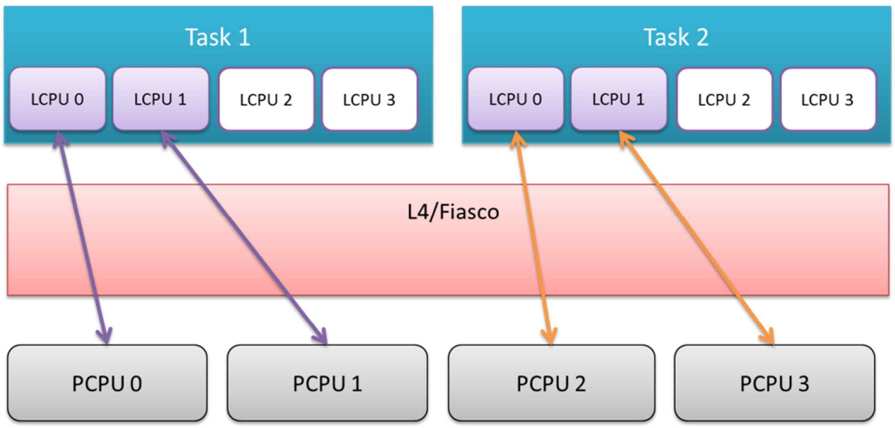

Figure 5-5: A scenario of L4/Fiasco with space-shared scheduler.

The emulation parameters are identical with table 5-1. The result from our evaluation is shown in Figure 5-6. In our scenario, all tasks are equally in high load. On the opposite, if one of the tasks is idle then the CPUs given to it will be not used optimally thus the whole system's performance is decreasing for space-sharing scheduling.

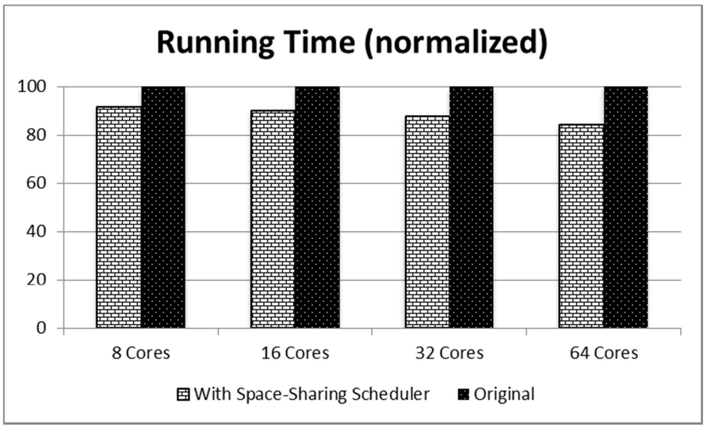

Figure 5-6: The normalized running time comparing L4/Fiasco with and without a space-shared scheduler.

L4/Fiasco is employing time-sharing scheduling if there is more than one thread on the CPU with a round-robin mechanism [17]. In time-sharing scheduling, there is more than one thread in a CPU. Hence the fiasco is time multiplexing the threads. Meanwhile, in space-sharing scheduling, since there is only one thread per CPU, time multiplexing the thread is not necessary. This is decreasing the OS complexity. Better usage of non-chip memory which is user-controlled rather than OS-dictated also improving the performance of space-sharing scheduler [2]. It has to be noted again that in Figure 5-6, we combine time-sharing and space-sharing scheduling, not solely space-sharing scheduling.

C. Space-Sharing Scheduling in L4Linux

In here we extended our space-shared scheduler into a Linux environment. Linux has been ported into L4/Fiasco under the name of L4/Linux. We evaluated our modification and expecting our performance increase from previous emulation is inherited by L4Linux. For the benchmark application we are using UnixBench. It tests several parameters that are highly related to the system performance benchmarking. The complete emulation parameters are summarized in table 5-2.

Figure 5-7 shows relation between the host machine and guest machine. ARM Versatile Express board is chose because it is the most similar with our i.MX board that is available on L4/Fiasco simulation. Both of them have the same CPU architecture. If we are using the real board, the procedure of debugging and analysis is shown in Figure 5-8. On using the real ARM board we are going to utilize a hardware debugger from Lauterbach called as Trace32.

Table 5-2 Simulation parameters for space-sharing scheduling evaluation

Parameter		**Value**
Simulation Tool		QEMU 0.14
L4Re Version		February 2013 release
L4Linux Kernel Version		Linux 3.7
Host	Machine	Intel Xeon® E5606
	Host CPU Architecture	x86 64 Bit
	Number of Processor	8
	Memory	12 GB
Guest	Machine	ARM Versatile Express
	CPU Architecture	ARM Cortex A-9
	Number of Cores	4
	Memory	256 MB
	Number of VMs	2
Real Board	Machine	FreeScale i.MX6 Quadcore
	CPU Architecture	ARM Cortex A-9
	Number of Cores	4
	Memory	1 GB
	Debugger	Lauterbach Trace32

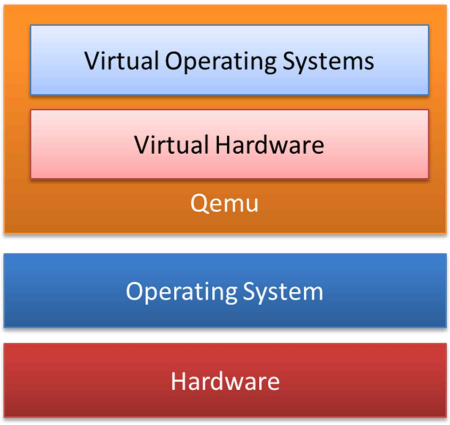

Figure 5-7: Simulation Environment using QEMU.

On measuring the performance effects of space-shared scheduler on L4/Fiasco, we are going to use the scenario in Figure 5-9 against L4/Fiasco in Figure 5-10.

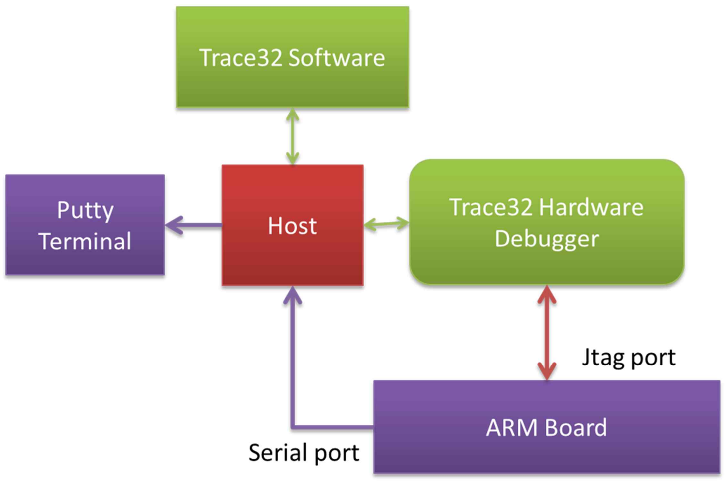

Figure 5-8: Simulation environment using an ARM Board.

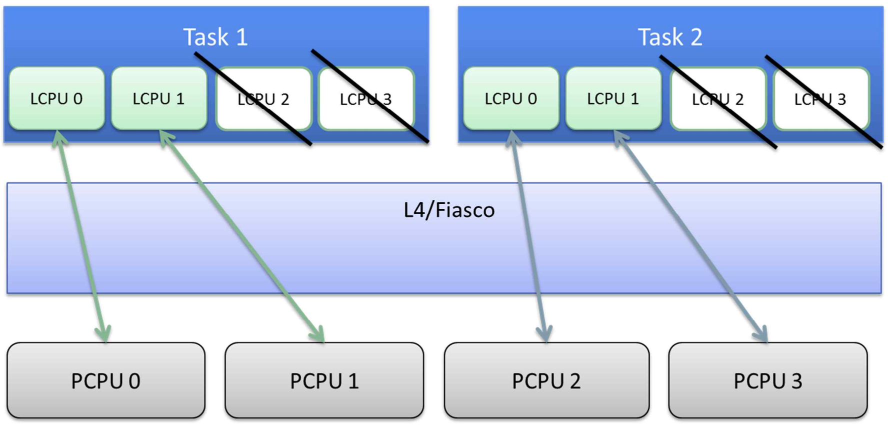

Figure 5-9: Simulation environment for fiasco with a space-shared scheduler.

Basically, fiasco without space-shared scheduler will allow all VMs to use all available PCPUs. On the other hand, fiasco with space-share scheduler tries to isolate PCPUs of one VM to another.

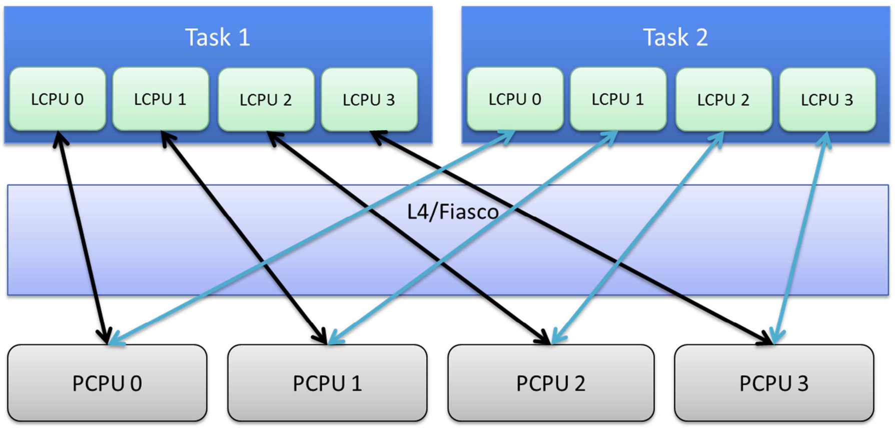

Figure 5-10: Simulation environment for fiasco without a space-shared scheduler.

D. UnixBench Parameters

In this thesis, we used UnixBench. The parameters that are used are explained in this subsection.

- **Dhrystone**

Reinhold Weicker developed Dhrystone in 1984. This benchmark is used to calculate and compare the performance of computers. It tries to characterize the result more meaningfully than MIPS (million instructions per second) because instruction count comparisons between different instruction sets (e.g. RISC vs. CISC) can confound simple comparisons.

- **Process Creation**

This assessment calculates the number of times a process can fork and reap a child that instantly exits. Process creation refers to essentially creating process control blocks and memory allocations for fresh processes, so this relates directly to memory bandwidth. Typically, this benchmark would be utilized to compare numerous applications of operating system process creation calls.

- **Pipe Throughput**

It is used to simulate the simplest form of communication between processes. Technically it is the number of times (per second) a process can write 512 bytes to a pipe before reading them back. However, in real-world programming, the pipe throughput actually test has no real equivalent applications.

- **Pipe-based Context Switching**

It is used to simulate the number of times that two processes can exchange an increasing integer through a pipe. Unlike pipe throughput, this simulation is relevant to a real world application. This benchmark spawns a child process with which it carries on a bi-directional pipe conversation.

- **File Operations**

The file operations test calculates the rate at which data can be transported from one file to another, using numerous buffer sizes. The file operations included in this test are file reading, file writing, and file copying. The tests capture the number of characters that can be operated in a specified time. In this simulation, we used two types of size; the big one and the small one. The sizes of each type are 1024 for buffer size, 2000 for maximum blocks and 256 buffer size and 500 maximum blocks respectively.

- **Shell Scripts**

This test measures the number of times per minute a process can simultaneously start and reap a set of various copies of a shell scripts. The script itself applies a series of transformation to a data file. In this test we create a set of one, two, four and eight copies of shell scripts.

- **System Call Overhead**

This measures the cost of entering and leaving the kernel, i.e. the overhead for executing a system call. It contains of a simple program constantly calling the getpid (which returns the process id of the calling process) system call. The time to perform such calls is used to calculate the overhead of entering and exiting the kernel.

E. Simulation Results and Discussion

In this section, we present the simulation results obtained and comparison of L4/Fiasco with space-shared scheduler against L4/Fiasco without a space-shared scheduler. Both are using L4Linux. The results of the simulation are shown in here.

- **File Operations**

  In Figure 5-11, we can observe how applying a space-shared scheduler will influence the file operation performance. From the graphic, we can conclude that the addition of the module increase the performance in file operations with the average increase around 4% until 7%. The small size slightly has better performance than the bigger one.

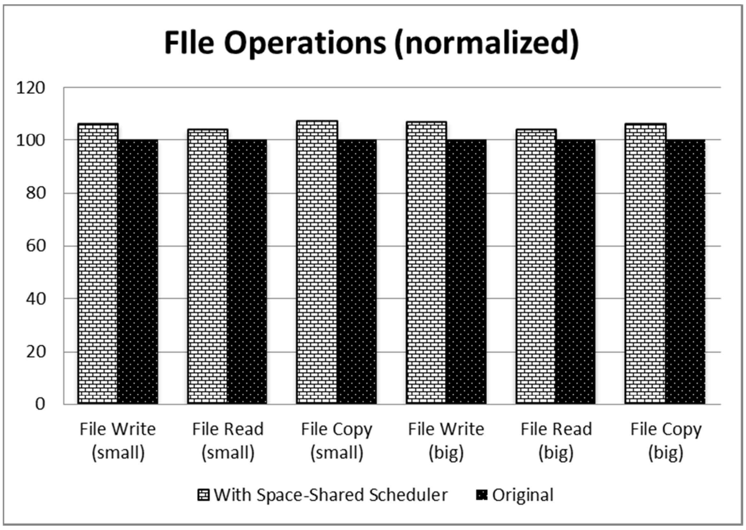

Figure 5-11: Performance operations in file operations.

- **Process Communication**

In process communication, we look how effective the communication through the pipe. In Figure 5-12, the results from pipe-based context switching and pipe throughput tests are displayed.

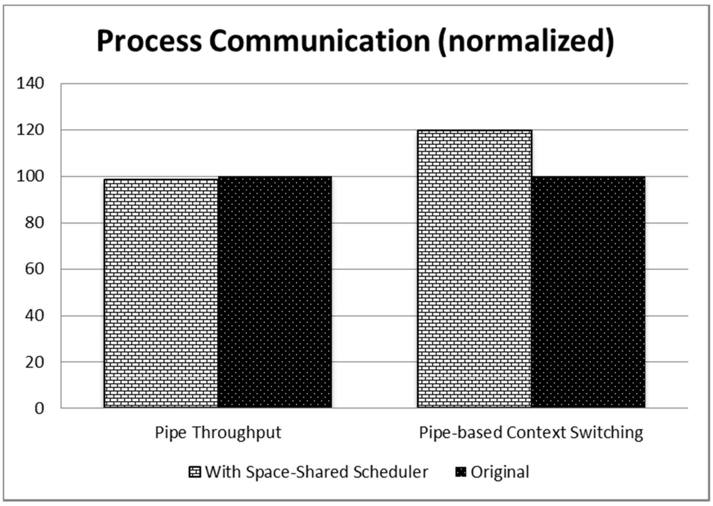

Figure 5-12: Performance operations in process communications.

- **Process Operations**

The result from the process operations is displayed in Figure 5-13. The small performance increase can be seen. The shell scripts tests are having performance increase when only 1 concurrent is used. One of our concerns is that when the number of concurrent process is increasing, as the characteristic of many-core systems, the performance is decreasing.

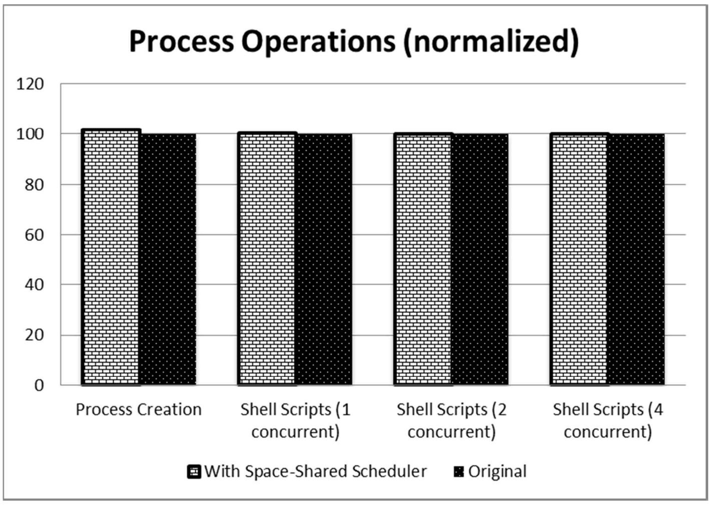

Figure 5-13: Performance operations in process operations.

- **System Overhead Call**

Using the space-shared scheduler, the cost of entering and leaving operating system kernel is reduced as can be seen by Figure 5-14. The *exec* operation has the most significant increase compared to others.

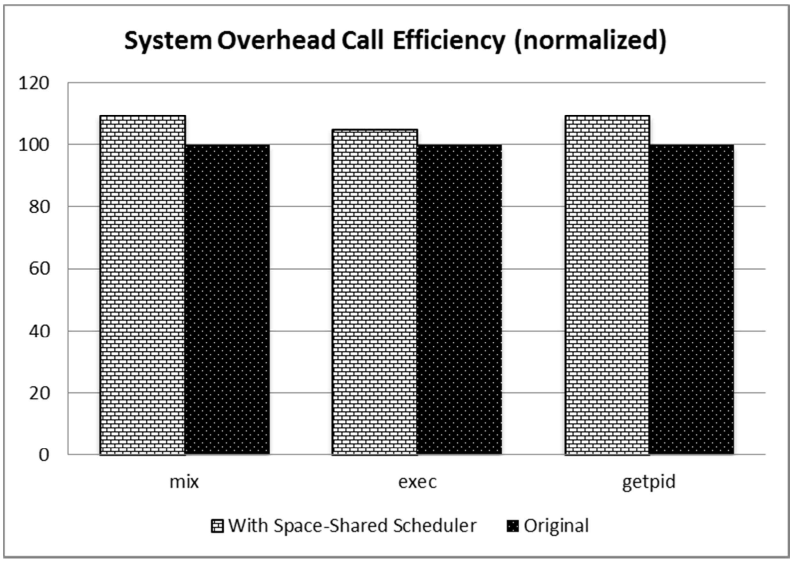

Figure 5-14: Performance operations in system overhead.

- **General Performance Test**

  In general tests, we conduct Dhrystone 2 and Hanoi Test. Both tests include some of operations from previous tests. The inherited performance gains from micro operations made these two tests show notable performances. The reason that the performance is increased while employing space-shared scheduler is suspected that cache invalidation in processors is decreasing as shown by our result in Figure 5-15.

F. Overhead in Implementing Space-Shared Scheduler

In here we are calculating the overhead that is gained due to the addition of space-shared scheduler. We vary the number of VMs from 1 to 10. The result of our evaluation is shown in Figure 5-15.

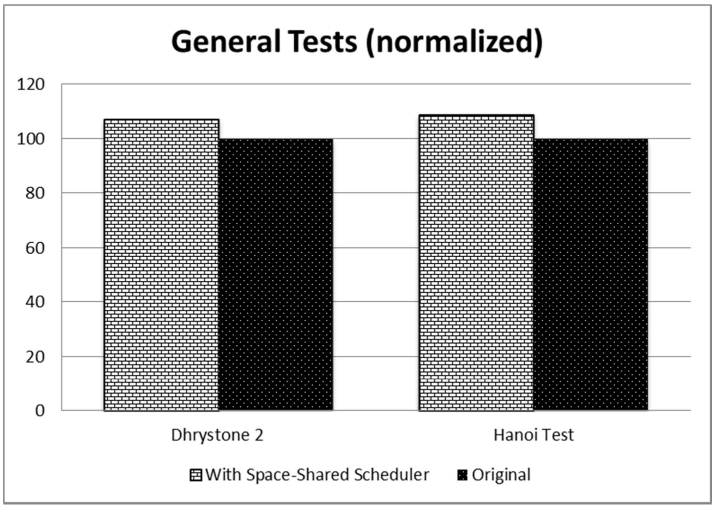

Figure 5-15: Performance operations in general tests.

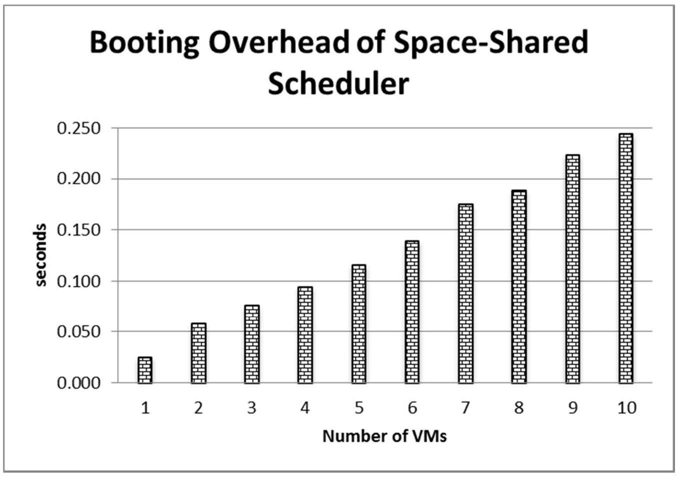

Figure 5-16: Overhead of implementing our space-shared scheduler.

VI. Conclusions and Future Works

In this thesis, we showed the current trends in processors architecture and the problems that may arise because of it from software perspective. The new approach to the scheduling design has also been discussed. Responding to these trends we argue that the addition of space-shared scheduler in Operating Systems is necessary.

We presented modifications to implement space-shared scheduler in L4/Fiasco. We benchmarked our modifications using CPU-intensive tasks and UnixBench to measure the performance improvements of our modifications. The scheduler automatically assigns CPUs to VMs. It tries to isolate the CPUs of one VM to another VM. Our simulation results show that the addition of space-shared scheduler into L4/Fiasco is beneficial as the performance outperform L4/Fiasco without a space-shared scheduler.

For future works, we want to run our modifications directly on our ARM board. The results are expected to be similar with the results from our simulations. Another future works, is to make the scheduler automatically able to adapt to the status of running VMs. For example, if one VM does not use given CPUs due to its load, the scheduler may give the CPUs to another VM with heavier load.

Bibliography

[1] Härtig, Hermann, Michael Hohmuth, and Jean Wolter. "Taming linux." Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART'98). 1998.

[2] Vajda, András. Programming many-core chips. Springer, 2011.

[3] Liu, Rose, et al. "Tessellation: Space-time partitioning in a manycore client OS." HotPar09, Berkeley, CA 3 (2009): 2009.

[4] byte-unixbench - A Unix benchmark suite - Google Project Hosting. [online] Available at: https://code.google.com/p/byte-unixbench/ [Accessed: 22 May 2013].

[5] Perumalla, Kalyan S. "μsik-a micro-kernel for parallel/distributed simulation systems." Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on. IEEE, 2005.

[6] Boyd-Wickizer, Silas, et al. "Corey: An operating system for many cores." Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. 2008.

[7] Schupbach, Adrian, et al. "Embracing diversity in the Barrelfish manycore operating system." Proceedings of the Workshop on Managed Many-Core Systems. 2008.

[8] Wentzlaff, David, and Anant Agarwal. "Factored operating systems (fos): the case for a scalable operating system for multicores." ACM SIGOPS Operating Systems Review 43.2 (2009): 76-85.

[9] torvalds (n.d.) linux. [online] Available at: https://github.com/torvalds/linux [Accessed: 30 May 2013].

[10] Ok-labs.com (2013) Home : Open Kernel Labs. [online] Available at: http://www.ok-labs.com/ [Accessed: 30 May 2013].

[11] Hartig, Hermann, et al. "The Nizza secure-system architecture." Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on. IEEE, 2005.

[12] Hartig, Hermann, et al. "DROPS: OS support for distributed multimedia applications." Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications. ACM, 1998.

[13] Os.inf.tu-dresden.de (2012) L4Re -- The L4 Runtime Environment. [online] Available at: http://os.inf.tu-dresden.de/L4Re/ [Accessed: 30 May 2013].

[14] Nightingale, Edmund B., et al. "Helios: heterogeneous multiprocessing with satellite kernels." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.

[15] The ARM Cortex-A9 Processors Manual.

[16] Qingguo, Zhou, et al. "A case study of microkernel for education." IT in Medicine & Education, 2009. ITIME'09. IEEE International Symposium on. Vol. 1. IEEE, 2009.

ABSTRACT

A Space-Shared Scheduler in Microkernel for Many-Core Systems

Ganis Zulfa Santoso
Advisor: Prof. Moonsoo Kang, Ph.D.
Department of Computer Engineering
Graduate School of Chosun University

The industry is running toward many-core systems. The increasing number of cores on a single chip introduced new issues such as decrease in memory bandwidth and less number of threads per core. To counter this, space-sharing scheduling paradigm has been proposed on the processor scheduling. The basic idea of such scheduling is to allocate a number of cores to an application and let the application manages its own resources.

Current Operating Systems are not employing space-sharing scheduling and also based on monolithic kernel. Time-sharing scheduling and monolithic kernel are not scalable for many-core systems. Another architecture of kernel called as microkernel gained attention lately because of its notable improvement in speed performance and its robustness.

This thesis implements a space shared scheduler inside a microkernel, L4/Fiasco, and performance will be evaluated using CPU-intensive tasks. The scheduler is extended and emulated in a Linux environment. The results show combination of space-sharing and time-sharing scheduling in L4/Fiasco is scalable for many-core systems.

ACKNOWLEDGEMENTS

I offer my sincerest gratitude to my supervisor Prof. Moonsoo Kang for his excellent guidance, caring and patience and providing me with comfortable research environment. His encouragement to pursue my master's degree at NAA (Network Architecture and Analysis Lab) and his continuous support have been a great assistance to achieve a milestone in my career. Without his help, this thesis would not have been completed. He is the best advisor that anyone could wish.

I would like to show appreciation to all members of the thesis examining committee Prof. Sangman Moh and Prof. Jaehong Shim for their treasured advices and insight. I would also like to express my gratitude to Dr. Joongsoo Lee, Dr. Youngwoo Jeong, and Timothy Anzaku from ETRI (Electronic and Telecommunication Research Institute) for valuable discussions and guidance. In addition, I would like to thank Department of Computer Engineering, Chosun University, to provide me the atmosphere to extend my knowledge.

My family in Indonesia has always supported me with their love. I would like to thank my mother Nenen Haryati and my father Dedi Soeprijadi, my brother Teguh Hadi Susilo, my sisters Popy Prihardani and Putik Seska Utami.

During my life in Korea, I have been supported by friends. I am thankful for Shelly Salim, Ivan Christian, Christian Oey, Irvanda Kurniadi, Adrianto Tejokusumo, Rico Hartono, Muhammad Bilal, Jeonghwa Park, Kelzang Tashi, Donggeun Cha, Junghun Kim, Jannies Lee, Karina Pramnanto, Youngim Kim, Donghun Kim, and others to be in my life.

Appendix

1. Code Modifications

As explained in the previous section, the modifications need to be done in three layers specifically in microkernel layer. We will explain the modifications along with the purpose in each layer.

A. Fiasco Microkernel Layer

In here the modification is done in [FIASO_DIR]/src/kern/scheduler.cpp. The objective in this layer is to create space shared scheduler object with the specification in the figure 2.

```
#ifndef CONFIG_NUM_VM
#define CONFIG_NUM_VM 1
```

Its objective is to take the configuration from the configuration file. If it is not defined, it sets to 1. The macro is defining how many VMs that will be used in this configuration.

```
class Space_shared
{
  unsigned vm_cnt; // counter to calculate VM
  unsigned based_num; // to encode the value to VM
  bool pairing_done; // indicate whether the pairing is
completed
  unsigned total_req; // calculate the total requested CPUs
  struct vm_t { // to record data for each VM
    unsigned id; // given ID to respected VM
    unsigned req; // requested CPU of respected VM
    unsigned cpu_sum; // given total number of CPUs to
respected VM
    unsigned long cpus; // value of given CPUs
  } vm[CONFIG_NUM_VM]; // create data as much as loaded VMs
};
```

Create Space_shared class along with its members. The explanation for each member is stated in comment next to it. Next, in the scheduler object we call the class in its class declaration by following line.

```
Space_shared ss;
```

On Operation enumeration, we add the following code. It is used so the abstraction layer in L4Re is going to use this code if it is communicating with the space shared scheduler object.

```
Req_cpu    = 3,
```

Adding the initialization method for Space_shared class.

```
PUBLIC void Space_shared::init()
{
  vm_cnt = 0;
  pairing_done = false;
  total_req = 0;
  int i;
  for (i = 0; i < CONFIG_NUM_VM; i++) {
    vm[i].req = 0;
    vm[i].cpus = 1;
    vm[i].cpu_sum = 1;
  }
}
```

Adding the method for Space_shared class if there is a request from VMs:

```
PRIVATE void Space_shared::add_to_db(unsigned req, unsigned
vm_id)
{
  if(vm[vm_id-1].req!=req)
  {
    vm[vm_id-1].id = vm_id;
    total_req-=vm[vm_id-1].req;
    vm[vm_id-1].req = req;
    total_req+=req;
    pairing_done = false;
  }
}
```

Adding method for Space_shared class if it is ready to pair VMs and PCPUs:

```
PRIVATE void Space_shared::pair(unsigned active_cpu)
{
  unsigned used_cpu = active_cpu;
  unsigned cpu_cnt = total_req;
  unsigned i, j, multiplier, x;
  unsigned long ret;
```

[Continued from above], set the values for each VMs.

```
  for (i = 0; i < CONFIG_NUM_VM; i++) {
    multiplier = 1;
    ret = 0;
    vm[i].cpu_sum = 0;
    for (j = 0; j < vm[i].req && j < active_cpu; j++) {
      x = cpu_cnt%used_cpu;
```

[Continued from above], value added 1 by default, so we can send PCPU 0. Since the architecture of the ARM board that we use is in serial shape, the pairing algorithm is simple.

```
      ret+=(1+x)*multiplier;
      multiplier*=based_num;
      --cpu_cnt;
      vm[i].cpu_sum++;
    }
    vm[i].cpus = ret;
  }
```

[Continued from above], set to true so we don’t need to run this again.

```
  pairing_done = true;
}
```

Adding method for Space_shared class to get the total of given CPUs for a VM:

```
PRIVATE unsigned long Space_shared::get_sum(unsigned vm_id)
{
  return vm[vm_id-1].cpu_sum;
}
```

Adding method for Space_shared class to get given CPUs for a VM:

```
PRIVATE unsigned long Space_shared::get_cpu(unsigned vm_id)
{
  return vm[vm_id-1].cpus;
}
```

Adding method for Space_shared class to process the request of a VM:

```
PUBLIC Utcb
Space_shared::req_cpu(Utcb const *iutcb, Utcb *outcb)
{
```

[Continued from above], get the values from utcb.

```
  unsigned req = iutcb->values[1];
  unsigned vm_id = iutcb->values[2];

  if(!vm_id) // if vm_id is 0, it means it has not been set
  {
    vm_id = ++vm_cnt; // get the new id
  }

  add_to_db(req, vm_id); // request! put it to db
  outcb->values[0] = vm_id; // return the id

  if(vm_cnt<CONFIG_NUM_VM)
  {
```

[Continued from above], if all VMs have not contacted Scheduler, it means the calling VMs have to wait.

```
    outcb->values[1] = 0; // not all vm requests are
received. Wait for more.
    return *outcb;
  }

  if(!pairing_done)
  {
```

[Continued from above], if all VMs have contacted Scheduler, and the pairing has not been done, call the the pairing function.

```
    unsigned i;
    unsigned active_cpu = 0;
    for (i = 0; i < Config::Max_num_cpus; ++i) {
```

[Continued from above], calculate how many CPU is online.

```
      if (Cpu::online(i))
        active_cpu++;
      else
        break;
    }
    based_num = active_cpu+1;
    pair(active_cpu);
  }
```

[Continued from above], if pairing has been done, no need to call pairing again.

```
  // cpus sets total
  outcb->values[1] = get_sum(vm_id);
  // cpu sets
  outcb->values[2] = get_cpu(vm_id);
  outcb->values[3] = based_num; // will be used to decode the
cpu_set

  return *outcb;
}
```

Call the initialization method of ss in the scheduler object:

```
ss.init();
```

Adding method for Scheduler class to call req_cpu function in Space_shared class:

```
PRIVATE
L4_msg_tag
Scheduler::sys_req_cpu(unsigned char, Syscall_frame *f,
                    Utcb const *iutcb, Utcb *outcb)
{
  *outcb = ss.req_cpu(iutcb, outcb);
  if(!outcb->values[0])
    return commit_result(0, 2);

  return commit_result(0, 4);
}
```

In kinvoke method of the Scheduler, we add this line of code:

```
case Req_cpu:    return sys_req_cpu(rights, f, iutcb, outcb);
```

B. L4 Runtime Environment Layer

The modification in this layer is basically just adding abstract functions to communicate with microkernel. The main function is:

```
L4_INLINE l4_msgtag_t
l4_scheduler_req_cpu_u(l4_cap_idx_t scheduler, l4_umword_t
req,
    l4_umword_t *cpu_set, l4_umword_t *cpu_set_sum,
    l4_umword_t *vm_id, l4_umword_t *mul, l4_utcb_t *utcb)
L4_NOTHROW
{
  l4_msg_regs_t *m = l4_utcb_mr_u(utcb);
  l4_msgtag_t res;
```

[Continued from above], the next lines of codes pass the message through IPC. `L4_SCHEDULER_REQ_CPU_OP` is set in the next line.

```
  m->mr[0] = L4_SCHEDULER_REQ_CPU_OP;
  m->mr[1] = req;
  m->mr[2] = *vm_id;
```

[Continued from above], the next lines of codes call the object of scheduler.

```
  res = l4_ipc_call(scheduler, utcb,
l4_msgtag(L4_PROTO_SCHEDULER, 3, 0, 0), L4_IPC_NEVER);

  if (l4_msgtag_has_error(res))
    return res;
```

[Continued from above], if there is no error, the return values set by the space shared object in microkernel will be used.

```
  *vm_id        = m->mr[0];
  *cpu_set_sum = m->mr[1];
  if(*cpu_set_sum)
  {
    *cpu_set     = m->mr[2];
    *mul          = m->mr[3];
  }

  return res;
}
```

Another modification is adding another operation code in L4_scheduler_ops enumeration. The value is needed to match with the operation code in scheduler object.

```
L4_SCHEDULER_REQ_CPU_OP = 3UL
```

C. Task Layer

The modifications in task layer are specific for each VMs, for example the modifications in L4Linux may be different from the modifications in another Operating Systems. Regardless of the differences, the modifications should be able to do these specific jobs:

- Inform space share module how much processors it needs,
- Acquire the physical id of CPU that is given by the space shared module, and
- Insert the values to the native mapping of physical and logical CPU.

Specific to L4Linux, these three jobs can be added in the *l4x_cpu_virt_phys_map_init()* function. We implemented the code that contacts L4Re in this function. L4Linux has native mapping of physical and logical CPUs, hence we do not need to create one and only added values that will be used by the mapping.

In *l4x_cpu_virt_phys_map_init()* function, instead of the native code that will only map the PCPU and LCPU from 0 up to the maximum number of maximal CPUs, we replace it with this code:

```
      unsigned cpu_set[NR_CPUS];
      vm_request = l4x_nr_cpus;
      l4_umword_t req = vm_request;
      l4_umword_t cpu_set_v, cpu_set_sum, mul;
      cpu_set_v = 0;

      // cpu_set_sum: total number of cpu given by the
scheduler
      // it can be bigger or smaller than the requested value

      //communicate with scheduler
      while(!cpu_set_v)
      {
        // get the value from scheduler
        l4_scheduler_req_cpu(l4re_env()->scheduler, req,
&cpu_set_v,
            &cpu_set_sum, &vm_id, &mul);
        //check if the requested is actually bigger
        if(req > mul-1)
        {
          req = mul-1; // substracted by 1 since mul is
always added 1 so we can pass cpu id 0
          vm_request = req;
          //request again if it is
          l4_scheduler_req_cpu(l4re_env()->scheduler, req,
&cpu_set_v,
              &cpu_set_sum, &vm_id, &mul);
        }
        if(!cpu_set_v)
          l4_sleep(100);
      }
```

[Continued from above], the value given by the Scheduler object is not a plain value, it needs to be decoded by following code. This is realized because the number of CPUs for each VMs may be different and it is difficult to allocate memory. Therefore the value is encoded. The mechanism will be explained in the section E.

```
      //decode the value
      unsigned div = 1;
      for (i = 0; i < cpu_set_sum-1; i++) {
        div*=mul;
      }
      unsigned cnt = 0;
      while(cpu_set_v)
      {
        cpu_set[cnt] = (cpu_set_v/div)-1;
        cpu_set_v = cpu_set_v%div;
        div/=mul;
        ++cnt;
      }
      // done decoded
```

```
// update according to the given value by the scheduler
l4x_nr_cpus = cpu_set_sum;

for (i = 0; i < cpu_set_sum; ++i) {
  if (i >= NR_CPUS) {
    LOG_printf("ERROR: vCPU%d out of bounds\n", i);
    return 1;
  }
  unsigned pcpu = cpu_set[i];
  if (!l4x_cpu_check_pcpu(pcpu, max_cpus)) {
    LOG_printf("ERROR: pCPU%d not found\n", pcpu);
    return 1;
  }
  l4x_cpu_physmap[i].phys_id = pcpu;
  for (j = 0; j < i; ++j) {
    overbooking |=
      l4x_cpu_physmap[j].phys_id == pcpu;
  }
}
```

2. Execution

A. Compiling

After the modifications, compile all the codes in all layers just like usual. Just don't forget to set the number of CPUs in kernel configuration.

B. Module List

1: entry L4Linux ARM with SS
2: roottask moe rom/l4lx_with_ss.cfg

```
3:  module l4re
4:  module ned
5:  module l4lx_with_ss.cfg
6:  module io
7:  module fb-drv
8:  module mag
9:  module vmlinuz.arm
10: module ramdisk-arm.rd
```

Line 2: set the root task to our configuration files.

Line 3-10: load all relevant modules.

C. Configuration Files

In this example, we are going to run two ARMs concurrently.

```
local lxname = "vmlinuz";
if L4.Info.arch() == "arm" then
  lxname = "vmlinuz.arm";
end

L4.default_loader:start(
  { caps = {
     shmns = shmns:mode("rw"),
    log = L4.Env.log:m("rws"),
     },
     l4re_dbg = L4.Dbg.Warn,
     log = { "l4lx 1", "yellow" },
 13:  },
 14:  "rom/" .. lxname .. " mem=64M console=ttyLv0 l4x_cpus=2
l4x_rd=rom/ramdisk-arm.rd root=1:0 ramdisk_size=4000 init/linuxrc");
 15:
 16: L4.default_loader:start(
 17:  { caps = {
 18:     shmns = shmns:mode("rw"),
 19:     log = L4.Env.log:m("rws"),
 20:     },
 21:     l4re_dbg = L4.Dbg.Warn,
 22:     log = { "l4lx 2", "RED" },
 23:  },
 24:   "rom/" .. lxname .. " mem=64M console=ttyLv0 l4x_cpus=2
l4x_rd=rom/ramdisk-arm.rd root=1:0 ramdisk_size=4000 init/linuxrc");
```

Line 16 – 24, is calling another instance of L4Linux ARM.
In line 14 and line 24, in the command line, we add another variable called as *l4x_cpus*. It is a native variable that is used to pass the number of CPUs for the respective VM.

D. Run using Qemu

```
qemu-system-arm -kernel bootstrap.elf -M vexpress-a9 -cpu cortex-a9 -
m 256 -smp 4 -serial stdio
```

-kernel: loaded kernel for the machine, adjust the directory to your kernel file.
-M: the machine that is going to be emulated

-cpu: the cpu architecture

- m: number of allocated memory (in MB)

-smp: number of cpu

-serial: redirect the serial port

E. Encoding Mechanisms

As stated before, the IDs of PCPUs that are given from the scheduler object to the tasks are not in form of arrays or lists, instead in a single number. To realize it, an encoding system is needed.

For example, if the scheduler gives PCPU with IDs of A_1, A_2,…, A_{n-1}, A_N. And there is an M number of CPUs in the machine. So the scheduler will encode the IDs with this method:

$$(1+ A_1) * P^0 + (1+A_2) * P^1+ \dots + (1+A_{N-1}) * P^{N-2} + (1+A_N) * P^{N-1}$$

With P = M + 1. The tasks receive the encoded code and decode it with the value M that is also passed. The ID is added by 1 to prepare if the scheduler wants to pass the value 0 with the P^{N-1}. We use the P instead of M so we can accommodate the addition in ID.

In a single number, the value can hold up to 15 CPUs at the worst case due to the limitation of memory.