Development of Multicore Computing for a Cloud-Based Unikernel Operating System

Maghsoud Morshed Chini Bolagh
Master’s Thesis Spring 2016
Development of Multicore Computing for a Cloud-Based Unikernel Operating System

Maghsoud Morshed Chini Bolagh

18th May 2016
Abstract

The cloud computing is going to change the IT design methodologies. This infrastructure requires innovation in order to enhance efficiency and functionality. One of the key element in the could infrastructure is the operating system which manages application and services. Hence, the operating system customization specifically for the cloud computing can enhance efficiency, scalability, and functionality. The Unikernel operating system will be the next generation of cloud operating systems and introduces a generative mechanism for the information infrastructure in the cloud. Since the processor vendors produce new processors with more independents cores inside and the cloud infrastructure equipped with high-end hardware and massive computing power, therefore the Unikernel operating systems should use this high computing power. Hence, the multicore computing can be a requirement for Unikernel operating system in order to enhance the performance and efficiency. Indeed, this thesis developed multicore computing for a Unikernel operating system such that it utilized new approaches in order to deal with multicore computing challenges in the virtual environment. The development provides the Unikernel operating system with as many cores as the virtualization platform can allocate to a virtual machine. Therefore, the new solutions developed in order to deal with the race condition, shared data and task management in the multicore environment. Finally, the results demonstrated that the multicore computing for an IncludeOS Unikernel operating system achieved better performance than a regular operating systems such as Ubuntu. Hence, the multicore computing enhances the efficiency, performance, and functionality of the Unikernel operating systems. Consequently, Unikernel operating systems can compete with existing solutions as a cloud operating system.
Acknowledgements

I would like to express my sincere appreciation and gratitude to the following people:

- First of all, I would like to thanks my wife Raheleh for her encouragement and rich discussions during my study. Her support, quiet patience, and unwavering love were undeniably bedrock upon which the past four years of my life have been built.

- Hårek Haugerund for his supervision, patience, support and encouragement throughout this thesis. It is not often that one finds a supervisor that always finds the time for listening to little problems that unavoidably encountered during the research. His technical and editorial recommendation was essential to the completion of this thesis.

- Kyrre Begnum who has been a constant source of enthusiasm and encouragement during the two years of my Master program. The door of his office was always open whenever I had a question.

- Alfred Bratterud for his technical feedback, support and encouragement during this thesis.

- Paal E. Engelstad for his encouragement and support during my study. In addition, for giving the opportunity to be an assistant for his courses.

- Anis Yazidi for his feedback and encouragement during my thesis.

- I am deeply thankful to my family for their love, support, and sacrifices. Without them, I would not be able to achieve this.

- My classmates and friends for all the worthwhile discussion and great times that we have shared.
# Contents

1 Introduction ............................................. 1
   1.1 Motivation ........................................ 1
   1.2 Problem Statement ................................ 6
   1.3 Thesis Structure .................................. 9
   1.4 Summary ........................................... 9

2 Background ............................................. 10
   2.1 Cloud Computing .................................. 10
   2.2 Cloud Operating System ......................... 12
   2.3 Unikernel Operating System ..................... 13
   2.4 IncludeOS .......................................... 14
   2.5 Multicore Computing .............................. 16
      2.5.1 Multicore Versus Multitasking ............ 17
      2.5.2 History of Multicore Computing ............ 18
      2.5.3 Multiprocessor (MP) Configuration Table .... 19
      2.5.4 Advanced Configuration and Power Interface (ACPI) 19
      2.5.5 Model Specific Registers (MSRs) .......... 21
      2.5.6 CPUID Instruction .......................... 22
      2.5.7 Advanced Programmable Interrupt Controller (APIC) 24
         2.5.7.1 Discrete APIC ........................... 24
         2.5.7.2 Local APIC .............................. 24
      2.5.8 Processor Initialization ...................... 25
      2.5.9 Bootstrap and Application Processors ........ 26
      2.5.10 Multicore Initialization Consideration .... 26
      2.5.11 Multicore Initialization Protocol ........... 27
      2.5.12 Multicore Message Passing Technique ...... 27
      2.5.13 Multicore Shared Memory Technique ........ 28
   2.6 Related Works ...................................... 29
      2.6.1 OSv Project .................................. 29
      2.6.2 MirageOS ..................................... 31
   2.7 Summary ........................................... 31

3 Methodology ........................................... 32
   3.1 Research Method .................................. 32
   3.2 System Design ..................................... 34
      3.2.1 Virtualization Platform ..................... 35
      3.2.2 Programming Languages ..................... 36
List of Figures

2.1 Structure of ACPI description tables ........................... 20
3.1 The research methodology diagram ............................. 34
3.2 Conceptual system design ..................................... 36
3.3 MP Floating Pointer Structure .................................. 39
3.4 MP Configuration Table Header ................................. 41
3.5 Root System Description Pointer ............................... 43
3.6 Root System Description Table ................................. 44
3.7 IA32_APIC_BASE MSR ........................................ 46
3.8 Multicore system supporting Hyper-Threading Technology 48
3.9 Processor topology enumeration from initial APIC ID .... 49
3.10 Interrupt Command Register .................................. 54
4.1 MP floating pointer table Results .............................. 63
4.2 MP configuration table Results ................................ 65
4.3 Root system description pointer (RSDP) table results ....... 66
4.4 Root system description table (RSDT) results .............. 68
4.5 IA32_APIC_BASE MSR value on the BSP ................... 69
4.6 The Intel® Core™ i7-3632QM processor supported feature on ECX inside the QEMU virtual machine ............ 73
4.7 The Intel® Core™ i7-3632QM processor supported feature on EDX inside the QEMU virtual machine ............ 74
4.8 The ICR value while sending the INIT IPI to all APs ........ 80
List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Web operating systems</td>
</tr>
<tr>
<td>2.2</td>
<td>Unikernel operating systems</td>
</tr>
<tr>
<td>2.3</td>
<td>The CPUID leaves and information which they can return</td>
</tr>
<tr>
<td>3.1</td>
<td>Vendor identification string returned by CPUID instruction</td>
</tr>
<tr>
<td>3.2</td>
<td>Initial APIC ID hierarchy for hypothetical system with 2 physical processors which each processor provide three cores and each core supports two logical processors</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution time of 500000th prime number computation within different operating system with 36 cores</td>
</tr>
</tbody>
</table>
Listings

3.1 The Bash command to create a virtual machine ................. 36
3.2 The IncludeOS project in GitHub ................................. 56
4.1 Multicore computing for IncludeOS project in GitHub ....... 60
4.2 Program code to retrieve the address of EBDA .............. 62
4.3 MP floating pointer data structure .............................. 62
4.4 MP configuration table data structure ......................... 64
4.5 RSDP data structure ........................................... 66
4.6 RSDT data structure ........................................... 67
4.7 Read the IA32_APIC_BASE MSR ............................... 69
4.8 Determine whether CPUID instruction is supported ......... 70
4.9 CPUID instruction ............................................ 70
4.10 Intel vendor ID ............................................... 71
4.11 Program code in order to retrieve Vendor identification string 72
4.12 Program code to determine the Hyper-Threading Technology support .................. 73
4.13 Determine mask width of SMT_ID and Core_ID ............ 75
4.14 Determine number of processor’s cores ...................... 76
4.15 Processor Topology Hierarchy Mapping ....................... 77
4.16 ICR low write operation to issue INIT IPI .................. 80
4.17 Programming Delay .......................................... 81
4.18 ICR write operation in order to issue STARTUP IPI ..... 82
4.19 Makefile rule ................................................ 85
4.20 Makefile rules for self-configuration code .................. 86
A.1 Multicore computing service .................................. 124
A.2 Self-configuration code ....................................... 132
A.3 Computation of nth prime number .............................. 134
B.1 Run multiple single-core IncludeOS instances ............... 135
B.2 Measure the time of the nth prime number computation .... 135
B.3 Kill IncludeOS instances .................................... 136
Chapter 1

Introduction

This thesis will develop the multicore computing for the Unikernel operating system. This chapter describes the research motivation and exigency of Unikernel operating systems in the cloud computing. The chapter continues with the problem statement to describe the problems which this research would face during the development process.

1.1 Motivation

The cloud computing is going to change the IT design methodologies. Everything is going to be on the cloud in order to be accessible through different platforms without location constrains. The benefits of cloud computing are in terms of availability and cost reduction. Hence, the server keyword in an IT infrastructure is not an expensive element anymore. The physical servers are going to be migrated to the virtual machines in order to build IT infrastructure in a cost-effective manner.

The cloud computing acceptance is growing among the people and it will alternate the physical infrastructure in the near future. This growing demand for cloud computing necessitates the improvement of this technology in order to build cost effective and more flexible infrastructure. This leads the rearrangement of the cloud computing building blocks to reach an efficient infrastructure while reducing the cost of ownership.

The cloud computing popularity is growing among the end users and business tries to accept it as a basis for their services. It offers a new opportunity for business such that it builds a scalable, available, flexible and affordable IT infrastructure in a reasonable time scale. Each of these makes it an attractive choice for the IT consultants and business in order to reduce the management costs of IT infrastructure.

The paradigm is changing IT infrastructure to be an easy setup service which every business builds its own business services in the public cloud in a couple of simple steps. This makes the cloud computing even more
popular so that it requires much more resources to satisfy the growing trend. However, building the bigger data centers and introducing the new hardware technologies will not compete for the growing trend and the key components should be rearranged.

The cornerstone of cloud computing is virtualization, which has been proposed to utilize the hardware. The virtualization has intended to utilize the hardware such that it enhances productivity and generativity of an infrastructure and accordingly cut the cost of ownership in the IT infrastructure. The virtualization scheme has been growing too fast in recent years in which it changed the information infrastructure rules and became an important building block of IT infrastructure.

The new data centers have been built to support cloud computing trend. The powerful computing hardware has been built as well as new data centers to support this huge infrastructure. Even though the technology improved the productivity of the hardware and data centers but data capacity is going to explode the storage. The information infrastructures are excessively using the cloud in order to facilitate information exchange among different parties. Although the deduplication techniques improved in order to eliminate the duplicates copies of repeating data but there are many unused data which is duplicated in the cloud storage.

On the other hand, the cloud computing should be generative infrastructure in order to be a pervasive technology among the users in the big data era. An Information Infrastructure (II) would be successful whenever it is generative. According to the Henfridsson and Bygstad (2013), there are three mechanisms that explain the evolution of IIs and the interaction between these mechanisms explain the outcomes of an II:

• Innovation, a self-reinforcing mechanism that new services and products are created as infrastructure flexibility spawns recombination of resources

• Adoption, a self-reinforcing mechanism by which more users adopt to the infrastructure whenever usefulness of the infrastructure increases by the investment of more resources.

• Scaling, a self-reinforcing mechanism by which an infrastructure expands its boundaries as it attracts new collaborators by creating incentives for collaboration.

However, a generative information infrastructure should be generative in these three mechanisms.

One of the important generative mechanism is the innovation which affects the productivity of the system. Cloud computing as an important infrastructure needs to be more innovative. According to Hanseth et al (2014), there are three types of innovation related to information infrastructure, innovation of, in and on infrastructure. Innovation of is about to implement a new infrastructure such as re-engineering existing infrastructure. Innovation in, extends a new infrastructure such as replacing or modifying
existing elements of infrastructure without changing architecture, and innovation on extending an infrastructure by adding new modules on what exists.

The technology would be called adoptable since it introduces pervasive features. The cloud computing as a technology is not an exception and should be more adaptive as possible in order to be a successful infrastructure. Also, the ease of mastery will help the infrastructure to be more adaptable. It explains that how easy it is for many audiences to adopt and to adapt the technology, how much skill is needed to make use of its leverage for the tasks they care about, irrespective of whether the technology was designed with those tasks in mind[3].

The biggest feature in the cloud is the ability to scale. Since the scaling is an important factor in the cloud infrastructure then it helps the infrastructure to be more successful. There are three main scaling approaches called as Scale Up, Scale Out, and Scale Side-by-Side. Hence, the cloud infrastructure should take care of the scaling and its consequence while its impact would not be well known in the large environment. In the side-by-side scaling, the infrastructure should handle multiple systems with duplicate data for different environments. So, cloud infrastructure should introduce new features to speed up such a scaling in a cost effective manner.

Although the cloud computing is a successful trend, but it should be generative enough to continue it’s success. While the cloud computing consists of different building blocks then each of them should be generative in order to guarantee the success of cloud infrastructure.

In addition, the cloud computing could be considered as an assemblage. According to Delanda (2006), Assemblage theory refers to relations of interiority. This means that elements of a system are constituted by the relations they have with other parts of the system. On the other hand, the components of the systems have no independent existence. Assemblages are heterogeneous and they involved in stabilization and destabilization the whole system. Delanda presents that Assemblage Theory helps to introduce the information infrastructure as an assemblage with heterogeneous interacting components and information infrastructure as a continuous processes of stabilization and destabilization[4].

The cloud computing as an assemblage consists of heterogeneous components such that the components can play stabilization or destabilization role in the cloud computing. The hypervisor, hardware, server, operating system, and network can stabilize or destabilize the whole cloud computing assemblage.

However, many research efforts have been dedicated to security, network, hypervisors, and cloud aware hardware, but one of the main building blocks on the cloud computing would be an operating system such that it can stabilize or destabilize whole the assemblage. Hence, the cloud computing requires a cloud-based operating system which is generative and as an assemblage play a stabilizing role in the cloud success.
The operating system as a component, can affect SaaS and PaaS products directly in the cloud infrastructure. The operating system manages virtual hardware to build a virtual machine and deliver a service to the users. Therefore, it has a direct relation with generativity and efficiency of the cloud infrastructure so that support the cloud to be successful.

The most common operating systems have been built to be an all in one operating system. This methodology of the operating system design sacrificed the performance. On the other hand, the performance becomes more important to the new scheme of the cloud computing which uses the one machine per task. In this scheme, the whole operating system should service one specific application while it is built to be a multipurpose operating system. Therefore, this idea comes into mind that a customize operating system should be developed in order to host specific application.

The operating system customization conforms the cloud computing paradigm and will increase generativity and efficiency while stabilizing the cloud infrastructure. This customization should follow a virtual machine per task scheme in order to build a task based operating system which emphasis on performance and cost. One solution to achieve this goal is to develop a Unikernel operating system, which every service delivered as a module. The operating system kernel is the cornerstone and other services act as a module to add an extra feature to the core operating system.

This idea motivates developers in Høgskolen i Oslo og Akershus Norway to develop a lightweight Unikernel operating system named IncludeOS that customized for the cloud. This tiny operating system uses a modular structure to become a Unikernel operating system for cloud computing. The new service utilizes the IncludeOS Kernel to interact with virtual hardware. This leads the IncludeOS to be a lightweight, high-performance Unikernel operating system in comparison with common operating systems used in the cloud infrastructure.

The other motivation for IncludeOS is to reduce storage consumption of unnecessary services which reside in the common operating system kernel. The excessive amount of unnecessary services and drivers require extra storage that will increase the cost of ownership in the cloud infrastructure so that the excessive storage increase the cost of ownership. The IncludeOS structure allows a service provider to customize the operating system more easily and efficiently only by adding or removing modules from the IncludeOS kernel. This leads the IncludeOS to consume less storage in comparison to the current operating systems.

The IncludeOS introduces a new paradigm for Unikernel operating systems which customize the operating system per service in order to increase performance and deduct the cost. The customized operating system proposes the better energy management indirectly. The minimal operating system has fewer services in contrast to common operating systems which requires less computing power. Also, it reduces storage consumption by eliminating the unnecessary files and services so that it
reduces the number of the storage devices for the same amount of services and accordingly reduce the energy consumption. Actually, the operating system has an important role in the performance and cost of ownership in the system. In the case of the voluminous infrastructure such as cloud, the cost of ownership would be more sensible in comparison to the small sized infrastructures.

Indeed, the performance and efficiency of the operating system as an assemblage of cloud assemblage has a multi-dimensional impact over the cloud computing. The performance of an operating system is directly related to the performance of it’s assemblages. As long as the Unikernel operating system as a set of modules then each of modules suppose to be a low level assemblage. Thus, the efficiency of underlying assemblages in the cloud environment play a stabilizing role in the operating system.

Since the system administrators are responsible for performance and resources of an infrastructure thus they should manage the operating system in order to ensure the maximum performance and efficiency. Since the operation and developing are not isolated tasks any more thus the system administrators can collaborate into the operating system development. Indeed, the Unikernel modular architecture facilitates the development collaboration. On the other hand, the DevOps culture has increased the collaboration of system administrators and developers in order to develop more stable and efficient product.

One of the main parameters in the performance measurement is computing power. Basically, computing power is mainly measured by processor frequency and performance. However, other parameters have an effect on the performance other than the processor such as memory and access time to the permanent storage devices. however, the processor has a high impact on the performance while the vendors produce more powerful processors today.

Since the hardware engineers faced the high-frequency constraints in the processor layout design then another approach considered in order to produce power full processors. The new paradigm for processor design changed to add multiple low-frequency cores in one processor package. Actually, it gives more computing power in the form of multiple computing units. This trend starts a new processor design paradigm in order to produce more cores per processor package. While these multicore processors are defacto processors on the market then each operating system should take advantage of these.

Indeed, every device has a multicore processor in order to compute faster. The cloud computing is not distinct in this trend and should use multicore processors so that it increase the computing power while reducing the cost of ownership. Meanwhile, the multicore computing is a feature which every operating system should support in order to generative system.

Meantime, the cloud environment by its nature requires a customized operating system which can take advantage of the many virtual processors
that a hypervisor propose to the virtual machine. Although the hypervisor uses the multicore processor in order to assign virtual processors to the operating systems, then multicore computing has a direct impact on the cloud infrastructure performance. This impact is higher while the hypervisor can oversubscribe the cores among the virtual machines.

Following this trend, this thesis proposes a development of multicore computing for cloud-based Unikernel operating system. Here, the basis for this development is IncludeOS Unikernel operating system which requires supporting multicore computing feature.

1.2 Problem Statement

Every operating system tries to use the hardware efficiently in order to increase the performance. In contrast to operating systems, hardware technologies promote faster in order to increase computing power and performance in smaller dimension. The operating systems are a way behind to use these cutting edge hardware devices, so these resources will no be utilized while the operating system cannot adopt itself to take advantage of the hardware.

The hardware companies introduce more features in each product announcement. This competition has emphasized more on the processor production, which companies try to follow the Moor’s law to double the number of transistors in an integrated circuit [29]. This scheme challenges the researchers to start the paradigm shift from increasing the frequency in one integrated circuit to build multiple low frequency integrated circuit in one processor package. This paradigm shift increased processor computing power and introduced the term multicore processor. Now the multicore processors are common and more cores invoke more computing power. Meanwhile, the hardware vendors have started the production of manycore processors which necessitates the customization of the operating system in order to take advantage of this computing unit.

The virtualization infrastructure requires a customized operating system to increase performance and reduce cost. This motivates the essence of Unikernel operating systems such as IncludeOS which customized for virtual infrastructure. In the cloud, the current operating systems use the multicore computing feature in their Kernel to improve the computing power, but the IncludeOS as an Unikernel operating system is a single core.

The system administrators are responsible for cloud infrastructure and accordingly they are responsible for performance and resource management. One of the common tasks of the system administrators is to manage the operating systems such that they configure, update and apply patches to the operating systems. The current operating systems have low efficiency such that they generate low output in regard to the resources they consume. Hence, the system administrators require productive and efficient
operating system in order to increase the performance and reduce the cost of ownership.

The Unikernel operating system promises the high performance and low resource consumption. The Unikernel has changed the operating system paradigm in order to build a simple and minimal operating system for cloud services. This will be a good news for system administrators so that they can be responsible for such an operating system which is simple and require less management time.

On the other hand, the DevOps culture invites the system administrators to collaborate on the development process such that utilize their operational experience in order to increase the performance of the operating system. Indeed, the performance tuning is one of the day to day system administrator task.

The Operating system performance enhancement can be done through the expansion of processing units. Hence, the cloud infrastructure can allocate multiple processing units to the virtual machines such that it creates a multicore processor for the virtual machine. Although the hypervisor allocates a multiple processing units to the virtual machine, but it will no increase the performance unless the operating system utilizes processing units.

By using the multicore computing the operating system can simultaneously execute different tasks so that it enhance the performance. Therefore, the main problem is to develop multicore computing for a Unikernel operating system such that it can execute tasks simultaneously. Although the main problem is to develop the multicore computing but the more specific questions will be raised in order to direct the research to its aim.

Nevertheless, the multicore computing requires the operating system be aware of the multicore processor in the system so that it can utilize them. Hence, the operating system should identify the multicore processor system and accordingly utilize it. The multicore development requires knowledge about the processor and the operating system as well. Therefore, the first question in this research would be how to develop the multicore computing for a Unikernel operating system.

The above-mentioned research question initiates more question about the project. The hypervisor allocates the multiple processors to the virtual machine and accordingly it forms a multicore processor for the virtual machine. In the multicore development, the processor cores should be identified and labeled so that they can communicate with each other. Hence, the operating system requires identifying the processors core in the virtual machine so that it can manage. Indeed, this would be analogous to an organization with multiple employees. Therefore, another question will be how to identify multiple cores in the virtual machine.

The multiple independent units will introduce the new problem over the shared resource. Whenever and independent units require accessing to a
shared resource, they should race with each other over the shared resource so that a shared resource introduces a race condition among the units. Indeed, this is analogous to an organization in which multiple employees require to access to a single line of telephone. Therefore, the development should answer the race condition in the multicore operating system and this raises a question how a multicore operating system should deal with race conditions.

The processing units require a procedure in order to handle the tasks. They require a procedure in which determine what they should do with a multiple tasks at the same time. This is analogous to an organization in which the employees may receive multiple cases at the same time. The operating system should determine the procedure for task execution in the multicore environment. It is same as the single-core operating system but the multicore operating system should also address that how a multicore operating system should execute multiple tasks.

The processing units may require sharing their data with each other. The output of one processing unit may require being part of an input for another processing unit so that the multiple processing units can interact with each other to serve the request in the operating system. In the multicore operating system, it may required that multiple core share data with each other so that the producer and consumer cores cooperate with each other in task execution. Indeed, this is analogous to an organization in which the employees require to share their data with each other in order to handle a case. In the multicore operating system, the cores should use a technique to share tasks and execution results with each other. therefore, the question raises how multiple cores should share data in the operating system.

Since the multicore computing aims to enhance the performance thus there is another option to do so. Hence, the multiple single-core operating systems will also enhance the performance so that the same computing enhancement will achieve through running the multiple operating system in the cloud environment. Here, the final question raises such that how a multicore Unikernel operating system performs in competing with multiple single-core Unikernel operating system.

This research will compose solutions in order to develop a multicore computing for a cloud-based Unikernel operating system. The project will utilize the IncludeOS as an existing Unikernel operating system. The research will propose solutions to all of the stated problems during the development such that the proposed solutions lead to the research aim. The following questions represent the problem statement of the research so that the research will propose the solutions in order to answer this questions and accordingly fulfill the research aim.

- How to develop a multicore computing for a Unikernel operating system?
- How to identify multiple cores in the virtual machine?
• How should multicore operating system deal with race conditions?
• How should multicore operating system execute the multiple tasks?
• How should multiple cores share data in the operating system?
• How the multicore Unikernel operating system performs in comparison to multiple single-core Unikernel operating system?

1.3 Thesis Structure

This thesis inscribes the research process in specific chapters. The chapter 2 will overview the existing system while presents a background about the multicore computing and accordingly investigates the related work. In chapter 3, the research method and system design will be presented. The solution will also propose in this chapter and accordingly the testing and evaluation process will be discussed. The chapter 4 will develop the solutions and test them accordingly until the research aim fulfilled. In this chapter the series of experiments will be implemented as part of evaluation. The chapter 5 will analyze the research and accordingly the research results. The chapter 6 will discuss the research and its finding during the development so that it can compare the achievement with competing solutions. The chapter 7 will conclude the research achievements and summarize the whole project. Finally, the appendices will contain the programming codes and the scripts which will be used in the research.

1.4 Summary

This chapter opened an introductory discussion about the multicore computing for a cloud-based Unikernel operating system. Afterward, it argued the upcoming problems and questions that the research should answer during the development in order to fulfill the research aim.
Chapter 2

Background

This chapter presents the concept of cloud computing, cloud operating system, Unikernel operating system, and IncludeOS. Then, the concept of multicore computing and its related topics will be discussed in the multicore computing section. Later, the related research efforts about Unikernel operating system will be discussed.

2.1 Cloud Computing

The IT industry looks forward to a utility which reduces the software deployment cost. The new software development paradigm no longer tolerates costly hardware and human operators to deploy services to the public. This made a paradigm shift in order to scale computing resources on demand and provide a pay-as-you-go business model for customers. The term cloud computing refers to services over the Internet and the hardware infrastructure with underlying software in the data centers which provides those services. The service itself has been referred as Software as a Service (SaaS) and the data center hardware and software is the so-called Cloud [5]. The cloud computing is growing fast and the IT companies such as HP, IBM, Microsoft and VMware invest their research capabilities in this area to claim the bigger share of the cloud infrastructure market. At the other side Amazon, GoGrid, Google, and RackSpace try to gain market share of public cloud and encourage the enterprise to invest their infrastructure in the cloud. Undoubtedly, more individuals and businesses will leverage the cloud in order to scale up their infrastructure while cutting down the budget, as reported by International Data Corporation (IDC) that cloud computing services will approach $100 billion by 2016 and enjoy an annual growth rate of 26.4% which is five times the traditional IT industry [6][7].

The cloud computing trend is growing in recent years. The recent report from RightScale Cloud Portfolio Management company surveying the technical professionals shows that 93% of them adopting cloud in
which 83% using the public cloud and 63% using private clouds in the 2015 [8]. This shows that almost every enterprise move toward cloud computing as their IT infrastructure. The new paradigm shift makes cloud computing more precise to have different cloud categories. The most popular categories are software-as-a-service (SaaS), infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS). The vendors are blurring the line between these categories to create public cloud platforms that can satisfy the needs of cloud developers [9]. This exploding trend of cloud platforms encourages the enterprise to spend more budget on their data centers or make an optimized cloud platform.

The cloud computing began its life to make the IT efficient while it needs the optimization itself. There are different criteria for the cloud computing optimization includes automated service provisioning, virtual machine migration, energy management and storage management. In the automated service provisioning scheme, the service provider objective is to allocate and deallocate the resources to satisfy the Service Level Agreement (SLA) which includes predicting the number of instances to handle the customer demand and future demands [10]. The virtual machine migration approach has evolved from process migration techniques [11]. Recently, live migration of virtual machines causes the short downtime and migrating the entire operating system and all applications as one unit will avoid difficulties of process level approach [12]. The energy management approach will reduce the operating cost of cloud computing. In the USA, the data center electricity consumption is predicted to increase to roughly 140 billion kilowatt-hours annually by 2020 [13]. Therefore, the energy management will benefit both sides of the scenario which are the providers of the cloud computing and the consumers.

On the other hand, the rise of social media and Internet of Things (IoT) has produced an overwhelming flow of data. The need to store, process, and analyze a large amount of data has driven organizations and enterprises to adopt cloud computing. Data creation at a record rate referred to herein as big data. The big data have high impact in cloud computing so that designers have a design challenge in order to develop appropriate platforms for data analysis and update [14]. The Google File System (GFS) [15] and Hadoop File System (HDFS) [16] have introduced as new Internet scale file systems which are suitable for distributed storage structure.

The optimization of cloud computing can be done on each of these areas. Each of these approaches needs a different research effort to evolve cloud computing for ongoing trend. All of these approaches have characteristics which are included in the cloud operating system. It means that optimizing the cloud operating system may cover all of these issues. Therefore, the cloud needs a customized operating system such that it has been optimized for the virtual environment.
2.2 Cloud Operating System

A cloud operating system is a customized operating system operating within the cloud environments and it is also called a virtual operating system. These operating systems are same as the common operating system and built to manage resources. In contrast to regular operating systems, cloud operating systems do not manage the hardware resources explicitly and are based upon other software such as hypervisors. This turns out the cloud operating system to an interesting research effort which will be the dominant operating system in the future as part of the cloud computing [17].

The cloud operating system aims to provide an interface for developing and deploying highly scalable distributed applications on behalf of a large number of users, infinite processor, storage, and bandwidth provided by cloud infrastructure. Indeed, the cloud operating system is evolving in order to provide isolation techniques and simple programming abstractions. The emphasize is to provide a much stronger level of integration with network resources. While a traditional operating system is a piece of software implementation that manage the hardware devices in a computer, the cloud operating system is a set of distributed processes whose purpose is the management of the cloud resources [18].

However, depending on virtual infrastructure and cloud services, the functionality of the cloud operating system may vary. There are two different trends in the cloud operating system which is based on the different target market. First, the cloud operating system for the cloud servers which target the performance and cost of ownership. Second, the cloud operating system for end users, which is called Web OS and target mobility.

The growing efforts have made in order to release commercial and open source web-based operating systems. While the idea is not new, the mobility of the application has made it more interesting. The web OS goes beyond basic desktop functionality and it includes many of a traditional OS capabilities including the file system, file management, productivity and communication applications. As the web OS functions across platforms from any device with Internet access. Actually, this brings the concept of mobility into the operating systems. The interesting point of the web OS is that it does not include drivers for computer hardware. On the other side, Internet technologies are increasing in bandwidth, which enables the faster communication of applications and data via the Internet to and from Web OS. Currently, available Web OSs include, Fearsome Engine’s Zimdesk [www.zimdesk.com], WebShaka Inc.’s experimental YouOS [www.youos.com], open source eyeOS Project’s [www.eyeos.com], Sun Microsystems’ Secure Global Desktop (SGD, [www.sun.com/software/products/sgd/index.jsp], and Sapotek’s Desktoptwo [https://desktoptwo.com], Computadora.de [https://computadora.de], the Google Chromium OS [https://www.chromium.org/chromium-os], ZeroDesktop, Inc. ZeroPC [http://www.zeropc.com/], SilveOS...
The table 2.1 classify the Web operating system.

<table>
<thead>
<tr>
<th>OS</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zimdesk</td>
<td>Zimdesk</td>
</tr>
<tr>
<td>YouOS</td>
<td>WebShaka Inc</td>
</tr>
<tr>
<td>EyeOS</td>
<td>open source project</td>
</tr>
<tr>
<td>Sun Secure Global Desktop</td>
<td>Sun Microsystems</td>
</tr>
<tr>
<td>Desktoptwo</td>
<td>Sapotek</td>
</tr>
<tr>
<td>ZeroPC</td>
<td>ZeroDesktop</td>
</tr>
<tr>
<td>SilveOS</td>
<td>Open source project</td>
</tr>
<tr>
<td>Google Chromium OS</td>
<td>Google</td>
</tr>
<tr>
<td>Computadora</td>
<td>Computadora</td>
</tr>
</tbody>
</table>

Table 2.1: Web operating systems

On the other hand, there are cloud operating systems which target the cloud infrastructure as their platform. These operating systems aim to host a specific service for many users and follow the one virtual machine per task operating systems. Hence, the performance and the cost of ownership are major factors in such these operating systems. However, the cloud uses the traditional operating system as a server operating system which looks at the virtual resources as same as physical hardware. In the cloud, there are fewer device drivers and the most of the cloud servers have planned to host one service. Hence, the traditional operating system would over consume the resources and accordingly disservice the performance and increase the cost of ownership.

While the cloud computing getting popular and enterprises move their infrastructure toward cloud thus it requires a customized operating system in which it adapts to the cloud environment. The operating system should address the cloud problems and accordingly improve the existing service. The Unikernel operating systems promise to enhance the performance while addressing the existing problems. This type of operating systems developed to improve the cloud service and accordingly reduce the cost of ownership.

2.3 Unikernel Operating System

The regular operating system designed to run on the hardware and require a variety of hardware drivers. Also, these operating systems intended to be multipurpose in a multiuser environment. On the other hand, an Unikernel is a single-purpose and is not designed to run on the hardware. Then it will be much simpler in contrast to regular operating systems. Indeed, the Unikernel operating system is not designed for multipurpose environment and consequently, it is a lightweight and simple operating system.
On the other perspective, the developer of MirageOS [21] describes the Unikernels as specialized kernels which are written in a high-level language and act as individual software components [22].

The Unikernel operating systems are evolving and more research efforts dedicated to developing Unikernel operating system. Currently, there are several under development Unikernel operating system projects. Some of these projects are based on the Linux kernel and some of them developed from the scratch. the Unikernel project developed with different programming languages so that some of them are locked in with particular language characteristics. Although some of the projects developed from scratch but they follow the core functionality of the traditional operating systems. Each of these Unikernel operating systems has the specific characteristic. Although all the Unikernel operating systems support common cloud features but they support different features in overall. The table 2.2 demonstrates the Unikernel operating systems, developer, programming languages, and the announced year.

<table>
<thead>
<tr>
<th>OS</th>
<th>Unikernel OS Developer</th>
<th>Programming Language</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniOS[24]</td>
<td>University of Illinois at Chicago</td>
<td>C</td>
<td>2011</td>
</tr>
<tr>
<td>MirageOS[21]</td>
<td>MirageOS is a Xen and Linux Foundation</td>
<td>OCaml</td>
<td>2013</td>
</tr>
<tr>
<td>OSV[25]</td>
<td>Cloudius System Project</td>
<td>C++</td>
<td>2013</td>
</tr>
<tr>
<td>ClickOS[26]</td>
<td>NEC labs</td>
<td>C++</td>
<td>2014</td>
</tr>
</tbody>
</table>

Table 2.2: Unikernel operating systems

The cloud operating system has developed by commercial, educational institutes, and enthusiast individuals. Following these efforts, research team in Høgskoken I Oslo og Akershus Norway has started a research project in order to develop an another Unikernel cloud-based operating system such that its target application will be web applications. Hence, it promises the enhanced performance and lightweight operating system. This Unikernel operating system named IncludeOS and has developed from the scratch using the C++ programming language.

2.4 IncludeOS

The cloud computing popularity has increased day by day and the educational institutes have also included it in the curriculum. The research
projects have started to invest on this field. As part of cloud computing investment in educational institutes, a Unikernel project has started in Høgskolen I Oslo og Akershus Norway which named IncludeOS in order to develop a lightweight operating system customized for the cloud.

This project aims to develop a lightweight operating system from scratch and implement traditional operating system concepts with a paradigm shift. In contrast to the traditional operating system, the IncludeOS aims to develop the necessary capabilities of an operating system and improve the performance by consideration of cloud structure. Hence, the surplus device drivers and many capabilities of traditional operating systems have neglected in order to improve performance.

The unnecessary piece of programs and files consume resources and make the operating system busy while managing them and this may affect the performance. In this project, the IncludeOS developed by the C++ programming language. As far as the C++ uses a direct mapping of hardware features provided by the C subset and has zero overhead abstraction on those hardware mapping then it will be a good choice to develop an operating system. Although, the C++ make efficient programs but the programming style and algorithm used is very important.

On the other hand, the C++ programming language has minimal output, which makes the whole operating system minimal as possible. The traditional operating systems have many unnecessary files and processes which consequently make the operating system big enough to occupy Gigabyte of storage. While the service itself occupies insignificant storage in comparison to the operating system. Hence, the cloud vendors should assign more budget for storage and maintenance of the cloud infrastructure. The installation of more storage appliances increases energy consumption and introduce new challenges in data center design. The cloud infrastructure consists of many assemblages which efficiency of each individual part affects the whole infrastructure and may stabilize or destabilize the whole infrastructure.

The IncludeOS project tries to develop such an operating system which address these problems in the ongoing trend of cloud computing. Hence, the IncludeOS image is lightweight and support UDP and basic TCP connection. The first service which IncludeOS equipped with is DNS such that the whole IncludeOS operating system consists of interrupt handlers, kernel, virtio-driver, network stack and bootloader occupies only 158 kilobyte [27]. As far as the UDP was the first network protocol which developed then the DNS as a UDP service is the first service of this project.

The IncludeOS has considered architectural changes in contrast to traditional operating systems which disabled the interrupt during the processing. The interrupts handling have an impact over the performance and waste CPU cycles in order to store and restore CPU registers. Although, the interrupt handling increases the responsiveness, but it has a performance impact.
In contrast to the most common operating systems, such as Linux or Microsoft Windows which follow the monolithic or hybrid kernel architecture, IncludeOS architecture is similar to Mach [28] microkernel. This allows to remove unnecessary services and build a minimal kernel which needs vital IRQ, drivers and process handlers.

On the other hand, IncludeOS focusing on the abstraction of underlying virtual hardware to make the service development easy. The modular nature of the IncludeOS allows the developer to change the kernel to make it compatible with the cloud service. This makes the IncludeOS a flexible operating system which uses the abstraction to hide the underlying hardware complexity and at the same time open the door for advanced developers to use the kernel core to make the customization.

The IncludeOS avoids using the hard disk to store data and memory paging. In other words, There is no virtual memory capability for it. This approach enables IncludeOS to remove the disk IO overhead to perform faster memory access. As far as IncludeOS is minimal then the whole operating system fits into the memory which eliminates the virtual memory usage.

This modularity is a part of the C programming language which makes it possible to include the kernel for a specific service. This makes the IncludeOS potential to be a multikernel operating system such as Barrelfish [29] which uses the multikernel architecture in order to manage many-core hardware. In contrast to the multikernel operating systems, the IncludeOS basically is single threaded and utilizes single core processor.

On the other hand, The IncludeOS has shortcoming as a modern operating system. It is only a single thread and has basic process management. As far as the cloud infrastructure equipped with cutting edge multicore hardware and allocates multicore processor to the virtual machines, then it emerges that the cloud operating system has the capability of multicore computing. However, the hardware vendors introduce the manycore processors in the near future therefore, the multicore computing will enhance the performance of the operating system.

This thesis as a part of this project aims, to develop multicore computing for the Unikernel operating system. The multicore computing will provide the IncludeOS much more computing power so that it can execute simultaneous tasks.

2.5 Multicore Computing

The computer users demand higher and higher computing power. This leads the Hardware manufacturers to increase the hardware frequency. The incrementing hardware frequency is notable on the processor chips, which is the symbol of computing power. The Gordon Moore predicted that the number of transistors per dens integrated circuit doubles approximately
every two years [30]. This massive growth of the transistors on the processor package introduce the new VLSI design challenges due to the high density of the transistors on the same space. The higher frequency consumes more power which presents a power wall problem. On the other hand, memory access latency is not in-line with the processor speed which presents the memory wall problem. Finally, the super-linear increase in design complexity without linear growth in the performance, which present the Instruction Level Parallelism (ILP) wall problem.

To overcome these problems, the frequency of the processor reduced and the number of the Integrated circuits per dies increased instead of increasing the number of transistors on the one circuit. Each core owns its cache to buffer the memory access and capable of doing an operation independently. This approach presented the multicore processor era to the computer world which breaks these walls to improve computing power. This finished the free lunch [31] and software are not able to use the higher computing power automatically. This solves the hardware design concerns and passes it to the software environment which require complexity to use these cores.

The multicore era began and introduced new architectures for computation. In these architectures, each core takes its own resources such as registers, execution units, cache and memory path which present new challenges. Now the Intel introduced more than 60 cores on its Phi coprocessor family [32]. Besides the challenges of using multicore parallelism, developers require a good understanding of buffer usage to avoid over-saturation of the memory controllers and the interconnections [33].

2.5.1 Multicore Versus Multitasking

Associated with the expanse of technology-based media in the human life, there is an ever-growing need for human to multitask. It is not surprising to hear that people describe multitasking as a ‘way of life’ [34]. These expectations flow in the computing world and people expect technology-based devices to operate multitasks simultaneously. These multitasking trends began from the first days of the computer life and now changed to the obvious capability of a computer system.

In the operating system design multitasking capability so-called multiprogramming. The multiprogramming present processor utilization which does more tasks in a certain amount of time. This cause the operating system handle more tasks which are likely might not wait for I/O at the same time. When one task needs to wait for the I/O requests, The processor can switch to the other task which is not waiting for I/O. This mechanism needs more memory to store the suspended programs until the I/O request finished. The modern operating systems benefit from this utilization.

On the other side, the multiprogramming has overheads which bring new complexity challenges. One of these complexities is memory management.
In the multiprogramming, the memory should hold the entire tasks to be ready after their I/O requests finished. This cause the processor to store its register value in the memory which is so called context switching. This overhead become worse when the memory has not enough space to store whole the programs state and uses the disk to store them.

The context switching is an act of storing the processor’s state of a thread and load the other thread. If the threads are associated with different virtual address spaces, then the content switch involves switching the address translation maps [35]. Depending on the processor cache system parameter, cache performance costs up to tens or hundreds of microseconds for an average context switch. So context switch wastes several thousand instruction cycles which are comparable to a network packet send and receive time. The cache-performance cost of the context switch may be the most notable among the other costs [36].

The context switching shares the processor among all programs and makes the multitasking available for multipurpose machines. This multitasking capability satisfies users with multiple different requests and makes the computers a multipurpose machine. On the other side, context switching introduces processor state management overhead, which may waste computation power for managing states.

The multicore computing may reduce these costs. In the multicore architecture, every core has its own cache and include a non-uniform memory architecture (NUMA) which every core exhibit non-uniform memory access times. This capability improves the multitasking and reduces the context switching costs.

Each core may use non-preemptive multitasking or preemptive multitasking on the operating system. Under the non-preemptive multitasking after resources allocated to a task, it cannot be interrupted until task finishes. This approach will remove the context switching overhead, but introduces a new problem when all resources are in use and new task requesting a resource. In this case, the new task may wait a long time, which causes the starvation problem and will not robust approach for the multipurpose systems. Under the preemptive multitasking, each core shares among the multiple tasks to utilize the resource management. Again, this will bring back the context switching overhead and cost.

The multicore processors enable the system to handle more tasks with lower overhead which improve the computing speed with the help of hardware computing instead of the operating system context switching.

### 2.5.2 History of Multicore Computing

In the October 1989 issue of the IEEE Spectrum, an article titled "Microprocessors Circa 2000" predicted that the microprocessor of 2000 could incorporate multiple processors with a cumulative performance of the 2000
million instruction per second [37]. In 2001, the first multicore processor called POWER introduced by IBM, which contains 170 million transistors and does 64-bit computation [38]. The multicore era continued by first Intel dual-core chip called Smithfield in 2004 which became one of the Intel’s top businesses and product initiatives [39]. At the time, AMD started work on a dual-core of its Opteron server processor, which it demonstrated in 2004 [40].

After all these inventions, it was time for operating system developers to change the scheme. The multicore operating system development became a research topic which still today is a hot topic. Now the challenges go beyond the multicore regular operating system and Unikernel operating system requires the multicore computing.

In order to develop multicore computing for an operating system, developers require to getting familiar with the multicore architecture. The operating system should retrieve the multicore processor in order to manage them. Hence, operating system require to retrieving multicore information through BIOS and the processor instructions. The multiprocessor specifications recommend the operating system to find the multiprocessor configuration table in order to develop the multicore computing.

2.5.3 Multiprocessor (MP) Configuration Table

The Operating System must have access to configuration information in a multiprocessor system. Following the system power-up, the BIOS detects installed hardware in the system and may create a structure to pass this information to the operating system. There are two such tables which contain the BIOS information called ACPI table and MP Configuration Table. The MP configuration table is an optional structure and may not exist. Then, the default information in the MP floating pointer structure should be used by the operating system. The software can find the MP configuration table address through MP floating pointer structure.

2.5.4 Advanced Configuration and Power Interface (ACPI)

The advanced configuration and power interface (ACPI) specification was developed to build industry common interfaces in order to enable the operating system to implement better device configuration and power management. The ACPI introduces better operating system-directed configuration and power management (OSPM). Indeed, ACPI evolved the existing pre-ACPI BIOS code, multiprocessor specification (MPS) tables, application programming interfaces (APIs, PNPBIOS APIs), advanced power management (APM) and so on into a robust configuration interface and power management specification. The ACPI supports the existing hardware and allows both legacy mechanism and ACPI exist in a single
machine. Therefore, operating systems should find and consume ACPI tables in order to get hardware information proactively [41].

The ACPI processor power management will save power in a working state and the OS may put the processors into low-power states (C1, C2 and C3) when the OS is in idle mode. The processor does not run any instruction in these low-states and should be awakened by an interrupt. The OS can read the ACPI power management timer in order to determine how much time has spent in the idle loop. It is notable that the power management timer runs at a known fixed frequency and OS can determine precisely its idle time. Then the OS will put the CPU into different low-power states. Also, the OS can use ACPI processor performance states in order to make a trade-off between performance and energy conservation [41].

Figure 2.1: Structure of ACPI description tables

The ACPI specifies a hardware register interface that ACPI-compatible OS can use them to control power management features of a machine. Thus, ACPI introduces tables to describe system information and features which are much more flexible than legacy implementations. These tables contain devices on the system board and their capability information. Then OS is capable of controlling system devices without knowledge on how these system controls are implemented. The ACPI tables are accessible through the root system description pointer (RSDP) structure which is inscribed in the system memory by the BIOS. This structure contains the address of extended system description table (XSDT) which points to other ACPI description tables. Figure 2.1 illustrates the structure of the ACPI description tables. All description tables start with identical ASCII string headers. Although RSDP is a root table, but extended system description table (XSDT) points also to other tables in the memory. Thus, the XSDT always points to fixed ACPI description table (FADT) first which contain information about hardware features. The ACPI description tables are as below [41].

- Root System Description Pointer (RSDP)
- System Description Table Header
- Root System Description Table (RSDT)
- Fixed ACPI Description Table (FADT)
• Firmware ACPI Control Structure (FACS)
• Differentiated System Description Table (DSDT)
• Secondary System Description Table (SSDT)
• Multiple APIC Description Table (MADT)
• Smart Battery Table (SBST)
• Extended System Description Table (XSDT)
• Embedded Controller Boot Resources Table (ECDT)
• System Locality Distance Information Table (SLIT)
• System Resource Affinity Table (SRAT)
• Corrected Platform Error Polling Table (CPEP)
• Maximum System Characteristics Table (MSCT)
• ACPI RAS Feature Table (RASF)
• Memory Power State Table (MPST)
• Platform Memory Topology Table (PMTT)
• Boot Graphics Resource Table (BGRT)
• Firmware Performance Data Table (FPDT)
• Generic Timer Description Table (GTDT)

2.5.5 Model Specific Registers (MSRs)

The model specific registers (MSRs) are included in most of the IA-32 and Intel 64 processor family. There are two main MSR groups called architectural and non-architectural MSRs. The architectural MSRs expected to be supported in future processors while non-architectural are not guaranteed to be supported.

MSRs are readable and writable through RDMSR and WRMSR instructions respectively. The RDMSR instruction reads the contents of a 64-bit model specific register (MSR) which the MSR register address must specify in the ECX register. Then RDMSR instruction returns the contents of MSR into EDX: EAX registers in which EDX loaded with high-order 32 bits of the MSR and EAX loaded with low-order 32 bits. It is important that this instruction must be executed in the operating system privilege level zero or real-address mode. Otherwise, it generates a general protection exception #GP(0).

The list of MSRs exist in the Intel® 64 and IA-32 architectures software developer’s manual chapter 35 [42].
2.5.6 CPUID Instruction

The processor vendors provide an increasingly sophisticated means which software can identify the features and capability of each processor. Hence, the identification mechanism has evolved in conjunction with the processor family evolution. The evolution of the processor identification is necessary because the computing market must be able to tune the processor functionality across processor generations and models with the capability of differing between features.

In the Intel processor families, Intel extended the processor signature identification into CPUID instruction as the processors evolved. The CPUID instruction provides processor signature and information about the features that processor supported and implemented on the Intel processor. Before CPUID, developers should write an algorithm to detect differences between different generations of processors. The algorithm would serve as much as CPUID instruction does [43]. This is a standard instruction that other vendors like AMD [44] support it in their products.

Since Intel486 family, Intel has provided a straightforward method for determining whether the processor’s internal architecture is able to execute the CPUID instruction. In this method, the instruction uses the ID flag in bit 21 of the EFLAGS register. If the software can change the value of this flag, then the CPUID instruction is executable. Hence, the POPF, POPFD, PUSHF, and PUSHFD instructions can be used to access the flags in EFLAGS register [45].

The CPUID gives much information about processor vendor, processor string name, number of cores, maximum logical processors in the system, cache characteristic, digital thermal sensor, power management parameter, performance monitor, virtual and physical address space and check the processor available features like SSE4.2, AVX, HTT. Therefore, it is a powerful instruction in order to identify processor and its supported features.

The CPUID instruction supports two set of functions that returns basic and extended processor information. The CPUID instruction takes no parameters in Assembly language and it implicitly uses the EAX register to determine the category of the returned information. In the more recent terminology of Intel, this is called CPUID leaf.

On the other word, different leaves of CPUID should be achieved by their respective number as an input to the EAX register while issuing CPUID instruction. Some of the CPUID leaves return basic values like processor features and others return extended information like processor topology enumeration. Table 2.3 presents a list of possible CPUID leaves and their return information. It is important to keep in mind that each processor supports specific CPUID leaves in respect to the different architecture used. Thus, its maximum number of CPUID leaves should be considered before try to retrieve information from the leaves. If the value entered for
CPUID.EAX is more than maximum leaves supported by the processor or the leaf is not supported on the processor then all registers return zero.

<table>
<thead>
<tr>
<th>CPUID leaves</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf 0-3</td>
<td>Basic CPUID information</td>
</tr>
<tr>
<td>Leaf 4</td>
<td>Deterministic cache parameters</td>
</tr>
<tr>
<td>Leaf 5</td>
<td>Monitor/Mwait</td>
</tr>
<tr>
<td>Leaf 6</td>
<td>Thermal and Power management</td>
</tr>
<tr>
<td>Leaf 7</td>
<td>Structured Extended Feature Flags Enumeration</td>
</tr>
<tr>
<td>Leaf 9</td>
<td>Direct cache access information</td>
</tr>
<tr>
<td>Leaf 10</td>
<td>Architectural performance monitoring</td>
</tr>
<tr>
<td>Leaf 11</td>
<td>Extended Topology Enumeration</td>
</tr>
<tr>
<td>Leaf 13</td>
<td>Processor Extended state Enumeration</td>
</tr>
<tr>
<td>Leaf 15</td>
<td>Platform QoS Monitoring Enumeration</td>
</tr>
<tr>
<td>Leaf 20</td>
<td>Intel Processor Trace Enumeration</td>
</tr>
<tr>
<td>Leaf 21</td>
<td>Time Stamp Counter/Core Crystal Clock</td>
</tr>
<tr>
<td>Leaf 22</td>
<td>Processor Frequency Information</td>
</tr>
<tr>
<td>Leaf 23</td>
<td>System-On-Chip Vendor Attribute Enumeration</td>
</tr>
<tr>
<td>Leaf 80000000H-80000008</td>
<td>Extended Function CPUID Information</td>
</tr>
</tbody>
</table>

Table 2.3: The CPUID leaves and information which they can return

The CPUID uses 4 registers include EAX, EBX, ECX and EDX in order to return processor information. The output of the CPUID instruction is fully dependent upon the content of the EAX register as input. This means that by placing different values in the EAX register and then call the CPUID instruction, then the CPUID instruction will return specific leaf upon different value resides in the EAX register. Moreover, some recent features of the processor require the ECX register participation in order to return sub-leaves information.

First of all, the CPUID should be called with CPUID.EAX=0 in order to acknowledge the highest leaf that processor supports so that it assures that CPUID will return reliable information. Further, the CPUID should be called with the most significant bit of EAX set in order to obtain extended information about the processor. The extended function information can be available by calling CPUID with setting EAX value equal and greater than 80000000H.

As presented in table 2.3, CPUID return approximately exhaustive processor information which all of them are not related to multicore computing. Thus, particular CPUID leaves which returns useful information about multicore capability will be used as part of multicore processor identifica-
tion algorithm.

2.5.7 Advanced Programmable Interrupt Controller (APIC)

The advanced programmable interrupt controller (APIC) was introduced by the Pentium processors into IA-32 processors. The APIC is based on a distributed architecture in which interrupt control is distributed between two basic functional units in the processor called the local unit and the I/O unit. These two units communicate with each other through a bus called the interrupt controller communications (ICC) bus. The APIC units are collectively responsible to deliver interrupts from source to destination through the processor bus. There are discrete and integrated APIC implementations in the system base on the Processor family. Therefore, each of APIC implementation has its characteristic so that they are responsible for interrupts in the system.

2.5.7.1 Discrete APIC

In the early processor generations, the APIC designed to be an independent unit known as 82489DX APIC. The discrete APIC in the multicore system does not react to the STARTUP IPI and ignore it. Thus, operating system must detect the version of APIC beforehand and this can be achieved by reading the APIC Version Register bit [7:0]. If the higher 4-bit version number in the Version Register are cleared, then the system uses discrete 82489DX APIC. If the version number is between 10h-15h then the system uses Integrated APIC and all other values are reserved.

2.5.7.2 Local APIC

In the Pentium processors, Intel integrated the APIC with the processor and called as local APIC. The local APIC can be enabled or disabled by the APICEN input through APIC_BASE MSR. If the local APIC is used then APIC must be initialized to Virtual Wire interrupt mode by BIOS during system initialization. In the multicore processors with Hyper-Threading Technology, all logical processors are identical and contain mutual local APIC. But application processors remain in halted until the operating system sends them INIT and SIPI respectively to bring them online.

The Intel introduced local APIC with Pentium processors and included in the recent processor families. The local APIC has two primary roles in the processor:

- Local APIC receives interrupts from the interrupt pins which can be from the internal source or an external I/O APIC. Then local APIC sends these interrupts to its logical processor.
• In the multicore processor, local APIC sends and receive inter-
  processor interrupt (IPI) messages to and from other logical pro-
  cessors through the system bus (IPI messages can be used to distrib-
  ute work among a group of processors or wake up the application
  processors).

On the other hand, the external I/O APIC is part of Intel chipset. Its
primary function is to receive interrupts from external sources and deliver
them to the local APIC of appropriate processors as interrupt messages.
This can help processors to share the interrupt processing load with other
processors in the system [42, 46]

Each local APIC contains a set of APIC registers and associated hardware.
The APIC registers placed in the memory and can be written and read from
using the "MOV" instruction. Basically, local APIC receives interrupts from
following sources.

• Locally connected I/O devices from local interrupt pins (LINT0 and
  LINT1).
• Externally connected I/O devices from I/O APIC.
• Inter-processor interrupts (IPI) in order to interrupt other processors.
• APIC time generated interrupts to implement programmed count.
• Performance monitoring counter interrupts to detect performance-
  monitoring counter overflow.
• Thermal sensor interrupts to detect tripped thermal sensor.
• APIC internal error interrupts for error handling.

2.5.8 Processor Initialization

Following the processor power-up or an assertion of the RESET pin, each
processor connected to system bus performs a hardware initialization and
an optional built-in self-test(BIST). However, a reset sets each processor’s
registers to a known state and place each processor into a real-mode
addressing. The reset invalidates the internal caches, translation lookaside
buffers(TLB) contents.

Following the processor power-up or reset, the state of the flags and other
registers set to predefined values. For example, The state of the control
register CR0 will set to 60000010h to place processor in the real-address
mode with paging disables. The default value of the flags and registers
after power-up are available in Intel® 64 and IA-32 Architectures Software

The BIST process is optional and hardware may skip it after power-up.
Following the BIST the EAX register will show the BIST status if the EAX
is cleared, then processor passes the BIST otherwise the nonzero value
indicates a fault. Therefore, software can report the error by checking the value of the EAX after power-up or reset whenever the BIST has requested.

Following a hardware reset, processor identification and revision information have stored in the EDX register. This information can also obtain by the CPUID instruction leaf 1. The version information that’s returned in EAX is equal with EDX after power-up, however CPUID gives the flexibility of checking the processor signature at any time.

2.5.9 Bootstrap and Application Processors

In the multicore environment, there are two classes of processors: the bootstrap processor (BSP) and the application processor (AP). Following the power-up or system reset in the multicore system, the hardware selects dynamically one of the processors on the system bus as a BSP while the remaining are treated as APs. As a part of the BSP initialization mechanism, the BSP flag is set in the IA32_APIC_BASE model specific register (MSR). This flag expresses that the processor has selected as a BSP and the rest of the processors in the environment are designated as APs. In the MP environment, the BSP is responsible for booting the operating system.

The BSP is also responsible to startup and shutdown APs and it is the last processor during the system shutdown. It is necessary that operating system keeps that BSP’s local APIC ID, so shutdown BSP after all AP processors. As far as the BSP is responsible for initializing the operating system then the APs would be in the following states:

- The APs have been avoided to execute the operating system.
- The APs are in halted mode with interrupts disabled. This means that AP’s local APICs respond only to INIT inter-processor interrupt (IPI).

According to Intel MP Specification [46], the operating system should determine whether the system conforms to the MP specification. This is a standard approach that can be done by other means on the newer systems. It is notable that in Intel 64 and IA-32 processors with Hyper-Threading Technology, each logical processor on the system bus treated as a separate processor. Thus, one of the logical processors is selected as BSP and the others are designated as APs.

2.5.10 Multicore Initialization Consideration

The multicore initialization procedure requires following consideration in the system:

- After power-up or reset, each processor examines its BSP flag to determine whether it should execute bootstrap code or wait for SIPI
• All devices in the system that are capable of issuing a interrupt to processors must be inhabited during the AP initialization process including INIT-SIPI-SIPI sequence.

2.5.11 Multicore Initialization Protocol

The multicore initialization Protocol would initialize the logical processors in the system bus. Thus during initialization, following operations are carried out:

• Based on the system topology, each logical processor is assigned a unique APIC ID.
• Each logical processor gets a unique arbitration priority.
• Each logical processor executes BIST.
• After BIST, the processors execute the BSP selection mechanism.
• BSP creates ACPI tables or/and MP table.
• BSP adds its APIC ID into ACPI table or/and MP table.

After selection of BSP, the operating system executes on the BSP. Then operating system should initialize the application processors (APs) on the system through the following protocol.

• BSP defines a counter and set it to one.
• BSP sends a broadcast or directed INIT IPI to the APs.
• BSP sends a broadcast or directed SIPI contain a vector to AP self-configuration code to the APs.

2.5.12 Multicore Message Passing Technique

The common technique for the using program in the middle of another program is calling it by name. This is a common function call which is not more than exchanging address of the arguments. In some cases, passing the address of the arguments to the other program is not applicable and require to send a message to the target program. This approach causes to copy arguments into the messages and sent to the receiver program which might be large data communication. This presents the message passing technique for communicating between the programs.

Because of the high cost of argument transfer through the message which might cause large data transfer, the message passing system was in focused on the distributed systems. An analogy of the message passing system is a web server and a web browser communication scheme.

Due to a trend for parallel programming in the early 80s, different message passing environment was developed. By early 1992 it was clear that the
authors of these message passing libraries duplicated each other’s efforts then agreed to implement a common standard for message passing. This introduced the message passing interface (MPI) standard library [47].

The message passing divided into two categories which are asynchronous and synchronous message passing. The synchronous message passing is less complex and used in the environment which both programs working at the same time. This implies that the receiver of the message should be ready for the message, otherwise the sender program will freeze until the receiver answers the message. Nevertheless, the asynchronous message passing is more complex due to the requirement of message bus or the message queue. In the asynchronous message passing program send the message to the message bus and continue its execution and receiver pick the message from the message bus and answer the message to the message bus again. The asynchronous message passing well suited the distributed system which might not be working at the same time.

The message passing is widely believed to be a scalable solution for manycore machines, the latency of the message transfers, seriously limits the performance of this model [48]. Despite the limitation, message passing model is highly compatible with the future applications. While each core has its own cache, the MPI simplifies the inter-core communication by defaulting the memory to be private for each core. The MPI implementation challenge is to provide developer an abstraction over the architecture changes [49].

In addition to the abstraction challenge, the message communication difficulties should be taken into account. The out of order message delivery cause cost extra overhead to retransmit the message if the in order delivery is not supported by the hardware. Also, the message integrity may make the implementation more challenging in contrast to shared memory technique.

The MPI is a scalable solution for manycore processor architecture. It avoids the complicated memory management and the unnecessary share memory access which leading to reduce power consumption. Nevertheless, the research efforts take place to optimize its higher efficiency.

### 2.5.13 Multicore Shared Memory Technique

Many parallel programming architectures were developed over the shared memory. For instance, a directory-based cache coherence was introduced in by Censier and Featrier (1978). While the shared memory is a more convenient programming paradigm and provide better support for many legacy programs, the shared memory approach was focused [51]. In this approach, the processing cores communicate with each other through the shared memory locations and subsequently reading them. This facilitates migration from a sequential model to a parallel programming.
The shared memory model can not scale when the number of cores increases, so in the high-performance computing area, shared memory has not been used since it does not scale to the high number of nodes, but a hybrid programming model answer this problem. To support this hybrid model multicore processors support a shared address space between cores and maintain a cache-coherent memory system. The cache coherency mechanism allows processors fast access of the commonly used data in their own private caches while maintaining the consistency when other cores update shared contents [52].

Due to cache coherency the data access locality optimization is more effective and easier with shared memory in comparison with the communication locality in the message passing technique [53]. The cache coherence has backward compatibility and this type of the distributed shared memory is a hardware-based model which is efficient on a small scale multicore systems [54]. On the larger scale processors, software level distributed shared memory can be used for higher performance. The first software distributed shared memory targeted single threaded computers. While the transition to multicore, the hardware evolved and involved in the software based model. In addition, emerging memory model like in C++11 demands new consistency for software distributed shared memory approach [55].

2.6 Related Works

The cloud computing is a hot topic and many researchers and developers invest their time in this area. Meanwhile, there are projects which focus on developing cloud-based Unikernel operating systems. These projects have different ambition and are developing features for their operating system kernel. However, there are two main cloud operating systems include Web OS and Unikernel operating systems. Hence, this thesis investigates the multicore computing for cloud-based Unikernel operating systems, thus the related work with the Unikernel operating system will be summarized in order to bring a good level of comparison.

2.6.1 OSv Project

The OSv is a Unikernel operating system released on September 16 2013, and its collaborators such as Glauber Costa, Avi Kivity, Pekka Enberg and Christoph Hellwig are well known in the Linux community. They are mostly known due to their role in creating KVM. It is a special purpose operating system which developed to run as a guest operating system on virtual machine [56]. The OSv is a minimal Unikernel including just the functionality necessary to run POSIX applications [57]. As a result, the OSv is not a multuser operating system or multiprocess while everything runs in the kernel address space [25]. The integration of both user and kernel address space removes the some of the extra operations associated with
IPC and context switching. The OSv uses large amounts of code from the FreeBSD operating system, especially the ZFS file system and the network stack. It is managed through the REST Management API and optional CLI written in Lua [58].

The approach taken by a cloud operating system such as OSv is to restructure the traditional approach to constructing the Library OS and customized it for cloud computing. The basic premise of this approach is to simplify the application stack, increase system security and removing layers of abstraction to reduce complexity [56]. Indeed, the OSv is designed to run under KVM, so it does not have a large set of device drivers. It is designed to run a single service, so a lot of the Unix-like mechanisms has been removed. The main difference with a common operating system is the integration between the kernel and user space. The OSv cuts a lot of the overhead associated with context switches; by forming a single address space, thus there is no need for TLB flushes, or to switch between page tables. Eliminating that overhead helps the OSv developers to reach lower latency than Linux [58].

The OSv proposes new design about handling multicore processors which does not use spinlock for critical sections. Indeed, the code running on multiple CPUs might read or write the same address, typical SMP operating systems use Spinlocks. One processor acquires the lock using an atomic test-and-set operation, and other processors stay in a busy-loop until the lock has been freed. When an OS runs on hardware system, a spinlock might waste a little amount of electricity. However, developers often use other more sophisticated techniques and try to reserve spinlocks for short-term situations.

But in the virtualization, the hypervisor might pause a virtual processor when the guest OS can’t predict the duration of spinning. Lock holder preemption indicates the situation when a VCPU is preempted inside the guest kernel and holding a spinlock. While the VCPU acquired the lock during the preemption, then any other VCPUs of the same guest that tries to acquire the lock must wait until the previous VCPU execute the code again and releases the lock. Lock holder preemption is possible if two or more VCPUs oversubscribe on a single CPU concurrently. Therefore, if more VCPUs in the guest run in parallel then more VCPUs must wait to acquire a lock. Hence, if the hypervisor pauses a virtual CPU while that VCPU holds a spinlock, it will be a bad situation where other virtual CPUs on the guest are just spinning, and guest–just waste electricity. Friebel and Biemüller (2009) indicate a solution to complain about the wait by involving a hypercall. But the OSv solution is to remove spinlocks from the guest OS entirely. Hence, the OSv does almost all of its kernel-level work in threads. Threads can use lock-based algorithms and are allowed to sleep. The threads in the OSv use mutexes, not spinlocks in order to protect the shared region. The mutex design is based on a lock-free algorithm by Gidenstam & Papatriantafilou (2009)[60].

One the other hand, the scheduler took control over threads switching.
Hence, the scheduler uses per-CPU run queues, so most of the scheduling operations do not require coordination among CPUs when a thread must move from one CPU to another. The lock-free design is one example in which OSv has tried to customize it as a Unikernel operating system for the cloud. The OSV designed to be a cloud aware operating system in order to prevent performance penalty and resource waste [61].

2.6.2 MirageOS

The Mirage Unikernel operating system, created by the Xen Project, while the developers are folk which developed Xen hypervisor. MirageOS has written from the scratch in a free and open source programming language called OCaml. MirageOS consists of functional implementations of protocols such as TCP/IP, SSH, HTTP, DNS, Openflow (switch/controller), XMPP and Xen inter-VM transports.

Mirage operates over Xen hypervisor and contains a few components which application should inherit from the operating system, such as a network stack. The applications are written in OCaml and deployed in a Unikernel style which communicates directly with Xen hypervisor through APIs. The Mirage developers claim that applications to their unique operating system operate faster than native or paravirtualized Linux. They believe that it is because of a clean and less complex design. MirageOS reforms the library OS concept, and tried to integrate the high-level programming, with low-level systems construction [21].

On the other hand, Mirage Unikernel is a completely new system and application must be rewritten from the scratch for this OS. Hence, the application developers must programme with Ocaml language. This would be a major challenge for developers since the vast majority of existing applications were not written in OCaml.

Indeed, at this point many of the use cases for Mirage seem quite focused and wouldn’t necessarily involve porting of common apps. Prototypes consist of hosting a high-performance web services running directly on the Xen hypervisor instead of a full operating system for administrators who can securely deploy network traffic monitors or isolated virtual switches into minimal virtual machines [62].

2.7 Summary

This chapter discussed the existing system and techniques in regard to research aim. Later, the related works investigated the related project aligned with this research.
Chapter 3

Methodology

This chapter presents a research methodology and illustrates the different parts of the research project including the system design, system components and methods of development in order to address the central research questions. Indeed, this chapter consists of research method, system design, development in order to develop multicore computing for cloud-based Unikernel operating system.

3.1 Research Method

Computer science is a new field to investigate computer as an object. However, computer science is based on mathematics and logic which both experimental and theoretical research methods follow the classical scientific pattern. The computer science is an ever developing field which evolves simultaneously with the development of theories. Hence, selecting a research method in computer science is challenging because benefits and challenges of using each method are not well listed yet. However, some of the methods follow the classical patterns while others are dedicated to computer science. Different methodologies have been proposed in computer science which some of them follow classical patterns and others are dedicated to the computer science. A research conducted in computer science can follow theoretical, modeling, simulation and experimental research methods. The computer science researchers may employ several methods in order to tackle the research questions [63].

The theoretical methods follow the classical logic and mathematical pattern in order to prove algorithms and systems. Indeed, this method follows a classical methodology by focusing on the mathematical aspect of computing in order to build theories as logical systems.

The modeling method is dedicated to creating a mathematical model. This model is usually based on a system in the real world and a model takes a specification of the system at the time $t$ then calculates the systems state
at the time $t+1$. On the other hand, the simulation method is based on the computer simulation of a model. A simulation is a program that uses step by step procedures in order to explore the behavior of a mathematical model. The modeling and simulation methods have a close connection together and a researcher may use both to tackle the research aim.

The experimental research methods are broadly employed in computer science in order to evaluate the new solutions. Indeed, the experimental methods are most effective on the projects that require complex software solutions such optimization problems, data management, and software development. The experimental method follows by experimental evaluation in order to appraise the experiment. The experimental evaluation usually includes exploratory and evaluation phases. In the exploratory phase, the researcher takes measurements in order to identify how to appraise the system and what question system should answer. Thus, the evaluation phase will assess the system and answer the questions.

Since this thesis investigates the multicore computing for the cloud-based Unikernel operating system, experimental research method provides the appropriate methodology to achieve the research aim. The development of multicore computing requires hardware level software development in order to manage processors. This complex procedure requires experimental methodology in order to explore hardware and software and consequently evaluate the solution recursively. On the other hand, the theoretical methodology focus on logical and mathematical pattern thus it can not be used to develop a software solution. The simulation methodology is based on a model to simulate a model of the real world system in order to investigate the system behaviour while the solution here will use an artifact. Hence, the experimental would be the proper methodology to achieve the research aim.

This thesis employs experimental research methodology as a scientific methodology. This logical scheme consists of different steps with iterative nature. The diagram 3.1 illustrates the logical structure of experimental methodology as the scientific method in order to develop multicore computing solution. As the diagram presents, the research is in a state of permanent development and change.

The logical scheme starts from existing system and explores related papers and resources. While the solution should employ the hardware, thus this phase generally based on exploring the hardware manuals and data sheets. Furthermore, the related literature and solution will be a source of acquiring knowledge to produce the new solutions. The logical schemes continue to solution phase which the new solution proposed in order to achieve the research aim. Since this is an experimental methodology thus many solutions can be generated such that solution would build upon earlier solutions. It brings the modularity to the project and will simplify implementation. Afterward, each solution should be implemented in order to be tested as an effective solution. If the implementation phase was successful then the solution should be tested on the system. On
the other hand, the implementation may lead to failure and may require more investigation in the exploration phase. The testing phase will test the implementation and determine whether the solution generates output as expected. If the implementation does not address the solution then it should be reviewed. Although the solution is implemented completely but it can not lead to research aim. In this situation, the development process should restart to exploration phase in order to gain better knowledge. Otherwise, the testing would be summit to the research aim.

Figure 3.1: The research methodology diagram

3.2 System Design

In each research, the architecture, components, modules, interfaces and data should be defined as a process of system design in order to address the research aim. The system design could be seen as building operational research methodology. The system will be designed with a modular architecture. The modularization refers to the logical partitioning of software structure so that the software become manageable during implementation and testing phases. The system design will be based on the existing IncludeOS structure. Therefore, the proposed system design will be restricted to the existing structure and should follow it.
The system design consists of following components:

- **Hypervisor**, the existing system uses the KVM (Kernel-based Virtual Machine) as a Linux kernel based hypervisor which uses CPU extensions (hardware assisted virtualization) via Linux kernel module. The KVM as a hypervisor, enables the system to run multiple virtual machines each has private virtualized hardware including virtual CPU, network adapter, disk and etc [64].

- **Emulator**, the existing system uses the QEMU (Quick Emulator) as generic machine emulator and virtualizer. The QEMU as an emulator, enables the operating system made for one hardware platform runs on different hardware platforms through dynamic translation. By executing the program code directly on the host CPU, QEMU can achieve near native performance [65].

- **IncludeOS**, the IncludeOS is an under development Unikernel operating system, which has developed with a modular structure. It is a single threaded which uses only one virtual CPU on top of the QEMU and KVM.

- **Service**, the service class is a module which enables the developers to contribute to the IncludeOS project. Developers can add their code in this module and build a new service for IncludeOS.

- **Programming languages**, the development process maps the solutions to program codes through programming languages. The development uses the C++ and Assembly programming languages. Indeed, the hardware level instruction will be coded through the Assembly language.

- **Compilers and linkers**, the whole development should finally be compiled to machine code and linked in order to build an executable image. The development requires also cross-compiler in order to produce a generic executable.

The figure illustrates the conceptual system design. The system design has a modular structure which follows a bottom-up structure. In this system, the KVM and QEMU components are interfaces between software and hardware. The IncludeOS with a modular structure is a component of the existing system and it hosts services. The new multicore service will be added to this system and programming languages will map solutions to program codes. The implementation and testing phases will be performed on this system in order to address the research aim.

### 3.2.1 Virtualization Platform

The virtualization platform consists usually of emulators and hypervisors in order to emulate the whole physical machine. The development process consists of recursive implementation and testing processes so
that virtualization platform facilitates the testing process on the same development machine.

The virtualization platform consists of KVM as hypervisor and QEMU as the emulator. While the KVM uses the hardware assisted virtualization, thus the processor should support hardware virtualization. The KVM package has been integrated into the kernel by kernel version 2.6.20 and higher. Therefore, The Ubuntu 14.04 has KVM up and running by default so there is no need to install KVM. The same happens to QEMU and it is integrated to the Linux Kernel. Therefore, the Ubuntu 14.04 has QEMU up and running by default and there is no need to install QEMU package. Hence, the command "kvm" will enable KVM and run qemu-system-x86_64 and -enable-kvm commands at the same time in order to build the virtual platform.

In order to prepare the system for multicore computing, the virtualization platform should provide the multicore environment for IncludeOS through determining the number of cores, threads, and sockets for the virtual machine. The virtualization platform should also do hardware assisted virtualization by delivering the host processor to the virtual machine. This can be done by the “-cpu host” option which delivers the host processor to the virtual machine. The listing 3.1 presents a Bash command in the Ubuntu 14.04 in order to build a virtual platform for IncludeOS operating system.

```
$KVM -cpu host -smp cores=36,threads=1,sockets=1 -nographic -hda IncludeOS_service.img -m 128
```

Listing 3.1: The Bash command to create a virtual machine

### 3.2.2 Programming Languages

Since this thesis is a development research thus programming languages are inevitable parts of the research. The C++ and the Assembly programming languages have chosen for development. While the whole IncludeOS project has developed through C++ and the author is familiar with C thus the C++ language is the dominant programming language during the re-
search. The multicore development requires interaction with hardware level thus C and C++ languages can not fulfill some implementations. Hence, the Assembly programming language is required in order to run processor supported instructions. It is notable that C and C++ language are able to include in-line Assembly in order to run hardware level instructions without optimization. But the multicore development requires building an boot loader for virtual processor initialization so that in-line Assembly can not address this condition.

3.2.3 Compiler and Linker

During software development, the program codes should be compiled and linked together into an executable. The C and C++ program codes can be compiled by Clang or GCC compilers. The existing IncludeOS system uses the Clang 3.6 as its primary compiler. Hence, this research uses the Clang as C and C++ compiler in order to follow homogeneous structure. The NASM will compile the Assembly program codes to binary.

All of the object files and symbol references should link together into an executable file so that virtualization platform boot the operating system. The existing IncludeOS system uses GNU LD as dominant linker tool so that this research uses it to follow homogeneous structure. The linker tool uses a linker script in order to form the memory layout of the operating system besides the location of objects in the memory.

The process of compile and link will be a cumbersome task in an iterative software development model. Hence, all the compile and link process will be automated with Make automation tool. The Make tool requires a Makefile which contains a set of software build directives. This file determines the process of making the executable by means of compiling program codes to object files and link them together.

3.3 Solution Development

Once the existing system design identified, the new solution will lead the research project to its destination. On the other hand, solutions should be inspired from validated sources in order to facilitate the software development. Since this thesis does research over developing multicore computing for IncludeOS operating system thus the solutions should concentrate on processor characteristics and how to use them in the process of developing multicore computing. Hence, processor manuals and data sheets would be the major source of information.

There are different processor vendors in the market and each of their product has different characteristics. But most of them follow almost the same concept with the different implementation. Hence, developing the multicore solution for one of the well-known processor families can contain
a broader group of users and at the same time facilitates the generalization of the development for other products. Since the Intel processor products are well-known and have comprehensive manual thus this thesis will develop the solution based on Intel processors. Therefore, the Intel® 64 and IA-32 Architectures Software Developer’s Manuals, MultiProcessor Specification, Detecting Multicore Processor Topology in an IA-32 Platform and Intel® 64 Architecture x2APIC Specification are the main sources during this research and heavily used in order to develop multicore computing solutions.

The following section will propose solutions in order to gather information about the processor and its supported features. Then, it continues with advanced programmable interrupt controller (APIC) and processor topology enumeration in order to use the processor interrupts. Later on, the multiprocessor initialization solution will lead to research aim.

3.3.1 Processor Basic Information

The first step in solution proposal in order to address the problem statement is to retrieve information about the processor architecture. This leads to investigate the multicore architecture and collect information to build a solution. According to the Intel MultiProcessor Specification, the MP configuration table is the first step to determine whether the system is a multiprocessor compliant. The address of MP configuration table can be retrieved through the MP floating pointer table so that the software should investigate it first.

3.3.1.1 MP Floating Pointer Structure

This structure contains an address pointer to the MP configuration table and other multiprocessor information. When this table is present then it indicates that the system conforms to multiprocessor specification. The Intel MultiProcessor Specification indicates that this structure must store at least at one of the following locations:

- Within the first kilobyte of extended BIOS data area (EBDA).
- In the last kilobyte of system base memory which usually can be in the 639 KB of base memory if the EBDA is undefined.
- If none of the previous location do not contain the structure then it should be in the BIOS ROM address space between 0F0000h and 0FFFFFFh.

The figure illustrates the MP floating pointer structure. The Multiprocessor compliant system must build the MP floating pointer structure with the minimum of 16 contiguous bytes and should be located in the high address of low memory address space. The contents of the structure must conform with following descriptions:
Figure 3.3: MP Floating Pointer Structure

- **SIGNATURE**, this is a 32 bit long ASCII string indicated by "_MP_" which can be used as a keyword to find the MP floating pointer structure location.

- **PHYSICAL ADDRESS POINTER**, This four bytes entry points to the beginning of the MP configuration table and it will be all zeros if the table does not exist.

- **LENGTH**, this one byte indicates the length of the floating pointer structure in the form of 16-byte (one paragraph) units. This means that if the table is only 16 bytes long then the value of length should be 01h.

- **SPEC_REV**, This one byte indicates the version number of multiprocessor specification in the system. Therefore, the value of 01h indicates version 1.1 and the value 04h represents version 1.4. The Intel MultiProcessor Specification is still supported for legacy MP capability while all new MP systems have been designed with ACPI in mind [66].

- **CHECKSUM**, This one byte indicates the checksum of the structure.

- **MP FEATURE INFORMATION BYTE 1**, if all bits of this one byte are zero, then the MP configuration table exists. Otherwise, the value of this byte represents the default configuration which is implemented by the operating system.

- **MP FEATURE INFORMATION BYTE 2**, The bit 7 of this one byte indicate the IMCR flag and the PIC interrupt mode is implemented if this bit is set. If this bit is cleared, then the virtual wire interrupts mode is implemented. All other bits in this byte [6:0] are reserved and all are zeros.

- **MP FEATURE INFORMATION BYTES 3-5**, these bytes are reserved and all are zeros.
It is notable that strings in the configuration table are coded in ASCII and the character order is from lowest byte to highest byte. Also, the strings are not null terminated and extra character locations are filled with a space character.

The software should use the SIGNATURE string as a keyword to find the MP floating pointer structure in order to retrieve the MP configuration table address. If the MP feature information byte 1 value is not zero and the physical address pointer is zero, then the MP configuration table does not exist and the operating system uses the default configurations. The software retrieves also information about the interrupt modes that the system use through IMCR flag resided in MP feature information byte 2 bits 7. The interrupt mode in the system can be PIC mode, virtual wire or symmetric I/O mode.

3.3.1.2 MP Configuration Table Header

The MP configuration table consists of a header called the base table and a series of variable length entries which follows it in increasing address [67]. The figure [3.4] illustrates MP configuration table header and it consists of following entries.

- **SIGNATURE**, this is a 32 bit long ASCII string indicated by "PCMP" which can be used as a keyword to confirms the presence of MP table.

- **BASE TABLE LENGTH**, This entry indicates the length of the base table in bytes.

- **SPEC_REV**, This one byte indicates the version number of multiprocessor specification in the system. Therefore, the value of 01h indicates version 1.1 and the value 04h represents version 1.4. The *Intel MP Specification* is still supported for legacy MP capability, while all new MP systems have been designed with ACPI in mind [66].

- **CHECKSUM**, this one byte indicates the checksum of the base table.

- **OEM ID**, This entry indicates a string to identify a manufacturer of the system.

- **PRODUCT ID**, this entry indicates a string to identify the product family.

- **OEM TABLE POINTER**, this entry point to an OEM-defined configuration table and it is optional. Therefore, if it is not present then this entry is zero.

- **OEM TABLE SIZE**, this entry indicates the size of the OEM-defined configuration table. Therefore, if it is not present then this entry is zero.

- **ENTRY COUNT**, this entry indicates the number of entries in the variable portion of the base table.
• ADDRESS OF LOCAL APIC, this entry indicates the base address of local APIC registers.

• EXTENDED TABLE LENGTH, this entry indicates the length of extended entries at the end of the base configuration table. If there is no extended entry therefore it is zero.

• EXTENDED TABLE CHECKSUM, this entry indicates the checksum for the extended entries.

The base configuration table contains entry count byte and it indicates the number of sub-tables after the base configuration table. These extra configuration entries after base table contain processors, buses and I/O APIC information in the system. For example, the processor entry can be retrieved in the 2Ch offset from the start of the base configuration table and contains the local APIC ID, local APIC version, BSP flag,
processor enable flag, processor signature and processor feature flags. While this information can be retrievable by the CPUID instruction then it is recommended to use the CPUID instruction to get this information. The complete information about the count entry sub-tables has stated in the Intel Multiprocessor Specification [46].

It is notable that, if the system does not update the MP configuration table then system initializes with default configuration. Thus, default configuration requires all local APIC IDs to be numbered sequentially starting from zero. The BSP must know about the APIC ID of APs in order to be able to send them directed IPI.

In case that the software could not retrieve necessary information about the multicore processor thus it can refer to the ACPI specification. The ACPI specification is created by BIOS and contains the system hardware information. Hence, the software can find the root system description table in order to investigate the ACPI specification in the system.

3.3.1.3 Root System Description Pointer

The OS must obtain the root system description table (RSDP) structure in order to locate the root system description table (RSDT) or Extended Root System Description Table (XSDT). Therefore, OS should search the memory for RSDP signature in order to find the RSDP structure on 16-byte boundaries. The RSDP should be in the following location in the memory.

- The first 1 kilobyte of the extended BIOS data area (EBDA).
- The BIOS memory address space between 0E0000h to 0FFFFFFh.

The RSDP locate usually same as MP configuration table in the BIOS memory space. Therefore, OS should search the memory space in order to find RSDP signature "RSD PTR". The figure 3.5 illustrates the RSDP structure. The RSDP table structure contains following entries that OS can use to access RSDT and XSDT.

- Signature, this 8 bytes entry is an ASCII string "RSD PTR".
- Checksum, this one-byte entry indicates the checksum of the first 20 bytes of the RSDP table.
- OEMID, this 6 bytes entry indicates the OEM-supplied string to identify the OEM.
- Revision, this one-byte entry indicates the revision of RSDP table. the larger revision numbers are backward compatible.
- RSDTAddress, this 4 bytes entry indicates the 32-bit physical address of RSDT.
- Length, This 4 bytes entry indicates the length of the table in bytes.
It is notable that reserved bits are all zero in the ACPI tables and the ACPI values are encoded in little-endian.

### 3.3.1.4 Root System Description Table

The software can locate the root system description table (RSDT) through the RSDT pointer in the RSDP structure. The RSDT structure shown in the figure [3.6] and it starts with a system description table header and followed by an array of physical pointer to other system description tables. The software should examine each table for a known signature. In regard to the signature, the software can interpret the information in the tables. The
software may find the MADT table physical address in the Entry section in order to retrieve information about the interrupts controller in the system.

<table>
<thead>
<tr>
<th>31</th>
<th>24 23</th>
<th>16 15</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Creator Revision</td>
<td>20h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Creator ID</td>
<td>1Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OEM Revision</td>
<td>18h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>space</th>
<th>D</th>
<th>I</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>B</td>
<td>A</td>
<td>T</td>
</tr>
<tr>
<td>space</td>
<td>D</td>
<td>I</td>
<td>M</td>
</tr>
<tr>
<td>E</td>
<td>O</td>
<td>Checksum</td>
<td>Revision</td>
</tr>
</tbody>
</table>

| LENGTH |
|--------|---|
| 08h    |

<table>
<thead>
<tr>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>T (54h)</td>
</tr>
<tr>
<td>D (44h)</td>
</tr>
<tr>
<td>S (53h)</td>
</tr>
<tr>
<td>R (52h)</td>
</tr>
</tbody>
</table>

Figure 3.6: Root System Description Table

The multiple APIC description table (MADT) describes all interrupts in the entire system so the operating system can use this table to get information about existing interrupt controller such as APIC, SAPIC or GIC in the system. The MADT structure contains the following entries.

- Signature, this 4 bytes entry is an ASCII string "APIC".
- Length, this 4 bytes entry indicates the length of the table in bytes.
- Revision, this one-byte entry indicates the revision of MADT table. the larger revision numbers are backward compatible.
- Checksum, this one-byte entry indicates the checksum entire MADT table that should be zero.
- OEMID, this 6 bytes entry indicates the OEM-supplied string to identify the OEM.
- OEM Table ID, this 8 bytes entry indicates the manufacturer model ID.
- OEM Revision, this 4 bytes entry indicates the revision of OEM table ID.
- Creator ID, this 4 bytes entry indicates vendor ID of the table creator.
- Creator Revision, this 4 bytes entry indicates revision of table creator.
- Local Interrupt Controller address, this 4 bytes entry indicates the 32-bit physical address of interrupt controller.
- Flags, this 4 bytes entry indicates the multiple APIC flags.
- Interrupt Controller Structure[n], this variable length entry indicates a list of interrupt controller structure which may contain all of I/O APIC, local APIC, I/O SAPIC, Local SAPIC, Non-maskable interrupt, interrupt source override, local APIC NMI source, local x2APIC and so on.

Another source of processor information and configuration are model specific registers (MSRs) such that they are readable and writable memory mapped registers. The MSRs are controller units which provide software in order to control the specific characteristics of the processor.

### 3.3.1.5 IA32_APIC_BASE MSR

One of the useful MSRs for the multicore system is IA32_APIC_BASE MSR which contains information about status of the local APIC. The figure 3.7 illustrates 64-bit IA32_APIC_BASE MSR contents and its bit fields. The bit fields are listed as follows.

- **BSP flag**, bit 8 indicates bootstrap processor (BSP) flag. This flag is set for the processor that selected as BSP during power-up or reset, otherwise it is cleared for application processors (APs).
- **APIC Global Enable flag**, bit 11 enable or disable the local APIC. While each logical processor has its local APIC then this flag apply to the relative processor.
- **APIC Base field**, bit 12 through 35 indicates the base address of the APIC registers. This 24-bit value is set to FEE0 0000 following by a power-up or reset.
- **Other bits are reserved and all are zero**.

In the multicore system, software should check the status of the local APIC in the IA32_APIC_BASE MSR in order to implement multicore initialization mechanism. Software should check the status of the current processor to know if it runs on the BSP or AP. Also, software can disable the local APIC by clearing the bit 11 but enabling the local APIC through setting the bit 11 require a soft reset. The APIC base address indicates
the start address of memory mapped APIC registers. The APIC registers are placed in an uncachable memory address and are heavily used in the multicore initialization protocol in order to send inter-processor interrupts (IPI) to other processors especially APs. Therefore, software should extract the APIC registers base address from IA32_APIC_BASE MSR in order to access the memory mapped registers.

In case that the MSRs do not contain the necessary information, then the CPUID instruction may return the operational information about the processor. The CPUID instruction can also return information about the processor vendor identification string.

### 3.3.1.6 Vendor Identification String

The basic application of the CPUID is identifying the processor’s vendor ID. This will help the software to decide the further procedures and the instruction set which may be applicable to this processor. If the EAX register contains an input value equal to zero, then the CPUID instruction will return the vendor identification string in the EBX, ECX and EDX registers. The table 3.1 presents the different vendor identification strings returned in the EBX, ECX and EDX registers when CPUID instruction with input EAX=0 is used [68, 69]:

If the CPUID instruction did not return any specific vendor ID, then further information of CPUID will not be reliable. For example, if the CPUID does not return 'GenuineIntel' string for Intel processors, therefore further CPUID return values for Intel processors will not be reliable.

The Intel processor can support the Hyper-Threading Technology so that it is a multicore processor. Hence, the software should determine whether the processor supports the Hyper-Threading Technology.

### 3.3.1.7 Hyper-Threading Technology

In 2002, Intel introduced Hyper-Threading Technology to improve the performance of processors when executing applications in multitasking
processors (AMD K5) | "AMDIsbetter!"
| VIA processors | "GenuineIntel"

old TRANSMETA processors | "TransmetaCPU"
| TRANSMETA processors | "GenuineTMx86"
| CYRIX processors | "CyrixInstead"

CENTAUR processors | "CentaurHauls"
| NEXGEN processors | "NexGenDriven"
| UMC processors | "UMC UMC UMC "
| SIS processors | "SiSiSiSi"
| NSC processors | "Geode by NSC"
| RISE processors | "RiseRiseRise"
| KVM hypervisor | "KVMKVMKVM"

Microsoft Hyper-V | "Microsoft Hv"
| VMware hypervisor | "VMwareVMware"

Xen hypervisor | "XenVMMXenVMM"

Table 3.1: Vendor identification string returned by CPUID instruction

environments. The technology enables each physical package to execute two or more separate code streams (called threads) concurrently. Later in 2005, Intel introduced IA-32 platforms with multicore technology in the Intel Pentium Extreme Edition processors. With this processor family, each package provided with two processor cores, each supporting Hyper-Threading Technology that shares execution resources. Thereafter, multiprocessing has evolved from multiple discrete sockets to Hyper-Threading Technology within multiple cores [70].

Each IA-32 processor that supports Hyper-Threading Technology contains two or more logical processors, which each of them has its own architectural state. This indicates that each logical processor has a full set of segment registers, data registers, debug registers, control registers, and most of the MSRs. Also, each of the logical processors has its own advanced programmable interrupt controller (APIC). This architecture allows two or more threads execute concurrently on each core. The core executes threads simultaneously, using out of order instruction scheduling in order to maximize resource productivity. [71]. The figure 3.8 illustrates a multicore system with two logical processors, each executes one thread simultaneously.

In accordance with Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A, processor name cannot present the support for Hyper-Threading Technology or Intel multicore technology. Therefore, the CPUID instruction should evaluate the processor capability [42].

The CPUID leaf 1 returns processor supported features in the ECX and EDX registers. Therefore, software can detect processor Hyper-
Threading Technology and multicore technology through CPUID.EAX=1 instruction. Thereafter, the bit 28 in the EDX register indicate the presence of Hyper-Threading Technology in the processor. Aligned with figure 3.8, the physical package is capable of supporting Intel Hyper-Threading Technology and/or multiple cores, if the EDX[28] is set.

CPUID.(EAX=1):EDX[28]=1

If the CPUID.(EAX=1):EDX[28] is cleared then the processor is single core and further multicore algorithm should be terminated. Otherwise, the algorithm will continue to enumerate the logical processors and physical cores within the processor.

The CPUID instruction contains the APIC ID of the logical processor in which execute the CPUID instruction. While the only available logical processor in the single core operating system is a bootstrap processor thus the CPUID returns the APIC ID of the bootstrap processor.

### 3.3.2 APIC ID

All of the APIC units require unique identification in order to be accessible through the other APIC units. Therefore, when a processor supporting Intel Hyper-Threading Technology is initialized at power up, each logical processor assigned with a local APIC ID. System hardware assigns a unique APIC ID to each local APIC unit on the system bus at power up. The local APIC ID can be also used as a logical processor ID in the Hyper-Threading Technology supported systems. Thus, software can use, local APIC ID which referred as initial APIC ID in order to communicate with other processors in the system [42].

The APIC ID assignment is based on system topology and includes encoding for threads, cores, sockets and cluster information. The initial APIC ID can also be used in order to determine the processor topology. The APIC ID can be detected by CPUID leaf 1 which returns supported feature
of the processor. Therefore, the bit 9 in the EDX register (CPUID. (EAX=1): EDX [9] =1) in the CPUID leaf 1 indicates that the processor contains APIC. Thereafter, the software can invoke CPUID in order to determine the initial APIC ID. Hence, software should invoke CPUID leaf 1 and determine initial APIC ID from EBX[31:24] register (CPUID. (EAX=1): EBX [31:24]).

![Figure 3.9: Processor topology enumeration from initial APIC ID](image)

The figure illustrates the processor topology enumeration through the initial APIC ID. The initial APIC ID may contain simultaneous multithreading ID (SMT_ID), physical core ID (Core_ID), package ID (Package_ID) and cluster ID. Each of these encodings should be determined before software can send an IPI message to other processors. Hence, the maximum logical processor ID and maximum core ID should be determined to enumerate processor topology from the initial APIC ID.

### 3.3.3 Logical Processor ID

The maximum number of addressable ID for logical processors in a physical package can be determined by CPUID instruction. This parameter is obtainable through CPUID while register EAX set to one and store the bits 16 to 23 of the EBX register (CPUID. (EAX=1): EBX [23:16]). The nearest power of 2 integers that is equal or greater than EBX [23:16] is the maximum number of unique APIC IDs reserved to address different logical processors in the package. It is important that if the processor does not support Hyper-Threading Technology then the value in EBX [16:23] will be zero. Though, software can treat this situation as a processor containing one logical processor per package.

On the other side, the maximum number of logical processors per physical core would be interesting in order to count the maximum core IDs available at the physical package. Hence, the maximum number of logical processor share the cache in one physical core obtained by executing CPUID leaf four while register EAX set to 4 and the ECX set to 0 the storing the return value of EAX [25:14] plus one (CPUID. (EAX=4, ECX=0): EAX [25:14]+1).
3.3.4 Processor Core ID

The second parameter in the initial APIC ID is the core ID. This parameter is determined by issuing the CPUID instruction with EAX equal 4 and the ECX equal 0, then storing the return value of EAX [31:26] plus one (CPUID. (EAX=4, ECX=0): EAX [31:26]+1). The nearest power of 2 integers that is equal or greater than EAX [31:26] is the maximum number of unique APIC IDs reserved to address different cores in the package. If the processor does not support the CPUID leaf four then the software should assume that this is a single core processor \(^\text{[70]}\). Indeed, this EAX [31:26] return value indicates only that the physical package can address this amount of cores and can not be used to determine enabled cores by the system BIOS.

3.3.5 Package ID

In the multiprocessor system, two or more socket may exist as physical processor package. The package ID sub-field identifies different physical processor within a cluster. The exact levels of hierarchical processor topology must be enumerated through CPUID leaf 0Bh. Common Intel processor families may employ the same topology represented by initial 8-bit APIC ID illustrated in figure 3.9. While CPUID leaf 0Bh can support topology algorithm that employs 32-bits x2APIC ID in order to decompose more than four sub-fields. It is notable that the width of each sub-field on figure 3.9 depends on hardware and software configurations. Hence, the software should map the processor topology hierarchy.

3.3.6 Hierarchy Mapping of Processor Topology

The CPUID leaf 0Bh identifies processor topology hierarchy in a deterministic manner. The hierarchy starts with the logical processor (SMT_ID) and depends on hardware and software configurations end up to package ID or cluster ID. The hierarchy mapping starts with 0Bh sub-leaf 0 (ECX=0) and incrementing the ECX value until the CPUID.(EAX=0Bh, ECX=n):ECX[15:8] returns an invalid level type encoding. This means that the ECX[15:8] will be equal input value if the level type is valid otherwise it would be zero (invalid). For example, if hierarchy does not contain Cluster ID then CPUID.(EAX=0Bh, ECX=3):ECX[15:8] will return zero indicating that there is no cluster in the processor topology hierarchy.

Along with ECX sub-leaf, the EAX register will provide enumeration parameters to extract width of each topology sub-field in an initial APIC ID. If the instruction CPUID.(EAX=0Bh, ECX=0):EAX[4:0] returns the width of the SMT_ID in the initial APIC ID, then return value in EAX[4:0] distinguishes the next higher topological entities above SMT level which would be the Core_ID. Therefore, for each subsequent high sub-leaf with index “n”, the CPUID.(EAX=0Bh, ECX=n):EAX[4:0] returns the width of
the sub-leaf in bits and allow software to distinguish the higher level topological entities.

The table 3.2 illustrates the initial APIC IDs for a hypothetical system with a dual processor which each processor provide three cores and each core supports Intel Hyper-Threading Technology.

<table>
<thead>
<tr>
<th>Initial APIC ID (8-bit)</th>
<th>Package ID (5-bit)</th>
<th>Core ID (2-bit)</th>
<th>SMT ID (1-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0h</td>
<td>0h</td>
<td>0h</td>
<td>0h</td>
</tr>
<tr>
<td>1h</td>
<td>0h</td>
<td>0h</td>
<td>1h</td>
</tr>
<tr>
<td>2h</td>
<td>0h</td>
<td>1h</td>
<td>0h</td>
</tr>
<tr>
<td>3h</td>
<td>0h</td>
<td>1h</td>
<td>1h</td>
</tr>
<tr>
<td>4h</td>
<td>0h</td>
<td>2h</td>
<td>0h</td>
</tr>
<tr>
<td>5h</td>
<td>0h</td>
<td>2h</td>
<td>1h</td>
</tr>
<tr>
<td>8h</td>
<td>1h</td>
<td>0h</td>
<td>0h</td>
</tr>
<tr>
<td>9h</td>
<td>1h</td>
<td>0h</td>
<td>1h</td>
</tr>
<tr>
<td>Ah</td>
<td>1h</td>
<td>1h</td>
<td>0h</td>
</tr>
<tr>
<td>Bh</td>
<td>1h</td>
<td>1h</td>
<td>1h</td>
</tr>
<tr>
<td>Ch</td>
<td>1h</td>
<td>2h</td>
<td>0h</td>
</tr>
<tr>
<td>Eh</td>
<td>1h</td>
<td>2h</td>
<td>1h</td>
</tr>
</tbody>
</table>

Table 3.2: Initial APIC ID hierarchy for hypothetical system with 2 physical processors which each processor provide three cores and each core supports two logical processors

The processor basic information helps to propose a solution to wake up the application processors in the system. Hence, the multicore initialization process will lead the research to wake up the application processors.

3.3.7 Multicore Initialization

The IA-32 architecture performs a multicore initialization protocol, which defines the boot protocol in order to initialize the IA-32 processors in a multicore system. The mechanism of performing multicore initialization protocol differs and is dependant to Intel processor generations.

3.3.7.1 Application Processor Initialization

The APs may be initialized by the BSP or by other active APs. The operating system may use the following universal algorithm to initialize the APs with a sequence of inter-processor interrupts. The inter-processor interrupts (IPIs) are issued by local APIC and require short programming delays in order to allow APs to execute IPIs. According to Intel MultiProcessor Specification [46], the operating system should use following sequence in order to wake-up the APs:

- BSP sends APs an INIT IPI
• BSP performs 10 mSec delay
• If the APIC is not 82489DX then BSP sends STARTUP IPI (SIPI)
• BSP performs 200 µSec delay
• BSP sends again the same SIPI
• BSP performs 200 µSec delay. The delay should be enough then all APs could initialize themselves and increment counter

Both INIT and SIPI are open-ended commands and operating system should determine whether the APs dispatched the SIPI. So the operating system should guarantee INIT or SIPI successful dispatch by checking the delivery status bit in the interrupt command register (ICR) and resending INIT or SIPI if the delivery status bit is one. Therefore, the operating system should poll the delivery status bit until successful dispatch. According to Intel MP Specification [46], the dispatch takes roughly 20 microseconds after that operating system should abort the command and resend it.

Also, the operating system can check the information provided by local APIC error status register. The local APIC status register bits [7:0] indicate different detected errors while initializing the APs. The local APIC status register consists following flags:
  • Illegal Register Address
  • Received Illegal Vector
  • Send Illegal Vector
  • Redirectable IPI
  • Receive Accept Error
  • Send Accept Error
  • Receive Checksum Error
  • Send Checksum Error

There is another set of important local APIC registers which enable the processor to trigger IPI toward other processors connected to system bus. The local APIC issues an IPI through interrupt command register (ICR). The ICR consists of two 32-bit memory location in the 300h and 310h offset from the local APIC base address.

3.3.7.2 Interrupt Command Registers

The interrupt command register (ICR) is a 64-bit local APIC register which allows the operating system to issue an IPI to another processor in the system bus. All the ICR fields can be read or written by operating system except delivery status field which is read-only. It is notable that writing to the low 32 bit of ICR trigger the IPI to target processors. Figure 3.10

52
illustrates 64-bit local APIC interrupt command register (ICR). The ICR consists of following fields:

- Vector, vectorized address of AP self-configuration code points 4-kilobyte boundary in low memory space.
- Delivery Mode, this 3 bits entry indicates the type of the IPI which is known as an IPI message field.
  - 000 (Fixed), this message indicates that local APIC delivers an interrupt specified in the vector field.
  - 001 (Lowest Priority), this message is same as fixed mode but the interrupt is delivered to a processor with the lowest priority among processors connected to the system bus.
  - 010 (SMI), this message delivers a system management interrupt (SMI) to target processor. The vector value should set to 00h for future compatibility.
  - 100 (NMI), this message delivers a non-maskable interrupt (NMI) to target processor. In this message, the vector information is ignored.
  - 101 (INIT), this message delivers initialization request to all processors. This message is sent to all processors regardless of the value in the destination field and the vector value is ignored. But the destination shorthand should be set to all excluding self mode.
  - 110 (Start-up), this message sends a STARTUP IPI to the target processor or processors. The vector points to the startup code which APs should execute.
- Destination Mode, this flag indicates the type of destination addressing which (0) indicate physical and (1) indicate logical destination mode.
- Delivery Status, this flag indicates IPI delivery status while (0) indicates local APIC completed sending IPI and (1) indicates local APIC has not completed sending IPI yet.
- Level, this flag indicates the level mode of the IPI signal. This flag should be cleared for de-assert mode and for other delivery modes should be set. It is notable that this flag has no meaning in Pentium 4 and Intel Xeon processors and must be always set.
- Trigger Mode, this flag indicates the INIT IPI de-assert delivery mode while (0) and it indicates activate with edge and (1) indicates activate with level.
- Destination Shorthand, this 2-bit field indicates the type of destination which local APIC should use to deliver IPI.
– 00 (No shorthand), this entry indicates that local APIC should read the destination address from 8-bit entry of ICR high double word [63:56].

– 01 (Self), this entry allow the local APIC to interrupt itself.

– 10 (All Including Self), this entry allows the local APIC to broadcast IPI in the system bus to all processor including itself.

– 11 (All Excluding Self), this entry allows local APIC to broadcast IPI in the system bus to all other processors.

• Destination, this 8-bits field indicates the target processor for IPI. This field is used only in conjunction with shorthand field set to 00h. In the physical mode, this field fills with the local APIC ID of the target processor.

3.3.7.3 INIT Inter-Processor Interrupt

An INIT IPI is an inter-processor interrupt to initialize the APs. The INIT IPI causes the processor to reset its state while it does not clear caches, floating point units and write buffers. The INIT must be used as part of warm-reset in order to prevent processor to run through entire BIOS initialization procedure (POST). This returns processor to a real-addressing mode.
The local APIC issue an IPI through interrupt command register (ICR) which are in the 300h and 310h offset from the local APIC base address.

### 3.3.7.4 STARTUP IPI

The STARTUP IPI is an interrupt with delivery mode "110" and trigger mode set to edge. The SIPI causes the target processor or processors to start executing in the real-addressing mode from 000VV000h address where the VV vector is part of the message. The startup vector is limited to a 4-kilobyte page boundary in the low memory space. Vectors A0-BF are reserved and cannot be used in the SIPI. The SIPI is not maskable and can be issued only one time after the INIT or RESET. So if the first SIPI received by the target processor then the other one do not cause to jump to vector again. The SIPI set CS: IP to VV00:0000h and the target processor execute self-configuration code at VV000 memory address. Indeed, the operating system uses SIPI in order to wake up the APs while cause them to jump immediately to the VV000h address and execute their self-configuration code.

It is notable that the operating system should not issue an SIPI to an 82489DX discrete APIC since the SIPI will be ignored by the 82489DX.

### 3.3.7.5 Self-Configuration Code

The self-configuration code is a piece of code that configures application processors to change the real-addressing mode to protected mode and make the processor ready for task execution. The self-configuration code usually known as trampoline code while it makes the processor jump to protected mode.

The main purpose of trampoline code is to initialize the segment registers and then jump to the protected mode in order to enable the processor to use high memory space. While the APs has no global descriptor table (GDT) and interrupt descriptor table (IDT) thus software may force APs to use the same GDT and IDT as BSP uses. Otherwise, the trampoline code should build a GDT and IDT for APs. While the trampoline code should be executed by APs, thus the whole trampoline code could be in a code section.

In the trampoline code, the application processors should increment a counter in order to announce that they are available to the operating system. The application processor must know about their GDT so that they know how to access the memory segments. The application processors must also feed with IDT so that they know how to handle interrupts.

Apart from the GDT and IDT, each application processor requires stack definition. Since there are limited general purpose registers thus application processor can not execute sophisticated tasks which require many
variables. Hence, the stack is a memory location for temporary data so that it will help the processor to execute sophisticated tasks such that require many variables. Therefore, each application processor should allocate dedicated memory location as a stack in order to execute tasks. Thereafter the application processor can jump to the protected mode and accordingly run in the kernel mode.

While the application processors fill their code segment and data segment registers with default kernel mode values thus they are available to the operating system for task execution.

At this point, the proposed multicore solutions should be translated to the program codes in the development phase so that the research move towards its aim.

### 3.4 Solution Implementation

The implementation part is actually where the solutions translated to the program codes and it would be discussed in the implementation chapter comprehensively. The solutions will be implemented by C/C++ and Assembly languages as a service for the existing IncludeOS Unikernel operating system.

Since an operating system is a complex piece of software which require version control thus the IncludeOS uses Git as its version control system. Therefore, the IncludeOS project is hosted in the distributed GitHub infrastructure in order to facilitate collaboration in the project. Each developer can fork the IncludeOS project and collaborate in the project. Thus, developers should clone their fork in order to develop locally in their system or developers can clone the IncludeOS project master branch directly. The listing 3.2 presents the IncludeOS operating project that can be accessed through GitHub.

```
1 https://github.com/hioa-cs/IncludeOS
```

Listing 3.2: The IncludeOS project in GitHub

After the IncludeOS operating system cloned thus IncludeOS operating system can be developed locally. This research will implement the multicore computing solution C/C++ program codes in the service.cpp file in the IncludeOs/seed/ directory accompanied by Assembly program codes in a separate file. The service.cpp file will compile and link to the IncludeOS operating system as a module in order to facilitate the testing part.
3.5 Testing

In this phase, the solution developer carries out testing in order to find out whether the solution is working as it should. The testing involves the execution of implemented solution in order to verify the following properties [72]:

- If the implementation meets its required results.
- If the implementation is stable with regards to all kind of inputs.
- If the implementation is usable.
- If the implementation achieves the project desire.

This phase consists of compiling the program codes and link it with existing IncludeOS operating system in order to build an executable image. Then, the image runs in the virtualized environment created by KVM/QEMU in order to test the implemented solutions.

The process of testing requires compiling the program code and link the multiple object files into a single executable image in an iterative procedure. This means that every testing process requires compiling and linking of multiple files into a single executable. If this procedure was manual then it will take a considerable amount of time in the development process. Therefore, this procedure should be automated through the Make automation tool.

The Make automation tool is a build automation tool which uses Makefile as its directive file. The Makefile contains a set of instructions with their dependencies. If none of the files that required by the linker have been changed then no compilation take place. This procedure can reduce the build time for large software project like this.

In the implementation chapter, the Makefile structure will be presented as part of testing phase in order to build an executable image file. After feeding the QEMU with IncludeOS image file, desired output can be tested through the IncludeOS output messages and the QEMU monitor console in order to track the characteristics and behaviours of processors and memory. Afterward, the development achieved its aim the research goes to evaluation phase in order to judge the solution.

3.6 Evaluation

The evaluation phase happens after the project achieved its goal in order to judge the developed solution in regards with other solutions. The evaluation phase presents the state of a particular developed solution based on the chosen criteria. Hence, the evaluation methods introduce actions which should be taken to measure chosen criteria. The evaluation may consist of following typical methods [73]:

57
• Analysis of data in order to find differences between the new solution and others.
• Observation of developed solution in actual use.
• Analysis of efficiency.

Among the typical evaluation methods, the analysis of efficiency will drastically fit the research aim. Since the IncludeOS is not in a production and there is no actual user for this operating system, thus analysis of data and observation of solution in actual use is not an applicable method for evaluating.

The software efficiency relates to the analysis of the time, cost, and materials that are expended by the solution in contrast to its effectiveness [74]. Hence, the evaluation phase that uses efficiency method should measure time, amount of memory and disk space usage by solution in contrast to other solutions. Here, the amount of memory and disk space is considered as the cost of a solution. Hence, the data collection through experiments helps the analysis of the efficiency and accordingly evaluate the developed solution.

3.6.1 Experiment

The evaluation phase can measure the criteria in a series of controlled experiments. Each experiment requires participants and here the participants are the multicore and multiple single core Unikernel solutions, bare metal Ubuntu 14.04 and Ubuntu 14.04 installed in a virtual machine. There are two major multiprocessor Unikernel solutions including achieving multiprocessor computing through building an operating system with multicore which this thesis concentrate on and building multiple single core operating system in a cluster.

Hence, the multicore Unikernel will be compared with multiple single core Unikernel, bare metal Ubuntu and Ubuntu installed in a virtual machine. Since this thesis is developing the solution in order to employ multicore processor in the Unikernel operating system thus it is also necessary to compare it with other multicore solutions.

However, the multiple single core operating system will use multiple memory and disk space units thus its obvious that the cost of running multiple single core operating system is more than one multicore operating system. This is due to the memory and disk space which should be duplicated through each operating system. Therefore, it is unnecessary to run the experiments in order to measure the cost of memory and disk usage among the multicore and multiple single core solutions. Otherwise, the total cost can be calculated by multiplying the $n$ number of tasks by the cost of the memory and disk. Hence, The only criteria that should be measured through experiments is the time of execution of a processor intensive task.
The cost of memory and disk in case of Ubuntu operating system is even higher while Ubuntu is a sophisticated operating system with various device drivers and services. Therefore, the Ubuntu requires much more memory and disk space in order to perform the same task in comparison with Unikernel operating systems.

The experiment will use a processor intensive task to load the processor and the execution time will be considered in order to evaluate solution efficiency. Developing a processor intensive task for the experiment is a challenging task which only loads the processor. There are many mathematical operations, which stress the processor and used in computer applications. One of these mathematical operations is the calculation of large prime number which has been used in the cryptography. On the other hand, the calculation of prime number is based on multiplication and division, which is handled by the processor. Hence, calculating prime number will be a good candidate for a processor intensive task.

The experiment can use prime number computing tasks with all participants in order to measure the execution time and evaluate efficiency. In order to collect valid data, the experiments should be repeated many times. The predictable output with limited variance in many experiments validates the collected data. Hence, 50 experiments will be enough for each solution in order to determine whether the collected data is predictable.

The results of the experiments will be considered as the main basis in order to evaluate the efficiency of the solutions. The upcoming chapters will discuss more about the experiments and process of data collection in order to evaluate solutions.

### 3.7 Summary

This chapter discussed the research method and architecture of this research project. The decision and solutions that have made in this chapter will be the cornerstone of the upcoming chapters in order to develop the solutions and evaluate them against the other solutions.
Chapter 4

Implementation

This chapter presents the development efforts of the proposed solutions which have been discussed in the previous chapter. Here, the solutions will be developed in the IncludeOS Unikernel operating system and the test results will be analyzed in the next chapter in order to achieve the appropriate outcome. This chapter begins with collecting information about multicore architecture available in the system. Later in multicore initialization section, the APs will be wakened up by issuing INIT-SIPI inter-processor interrupts. Finally, the experiments will be taken in order to evaluate the solutions with the competing one.

4.1 Build Existing System

This thesis is developing a new solution for existing IncludeOS Unikernel Operating system which is multicore computing. Hence, the existing system is a cornerstone of this research thus it should be built at the first step. Since the existing system has different collaborators and uses the Git version control system thus the whole project is accessible through GitHub web page. Each collaborator should fork the project and then clone its own fork in order to develop the project locally. The author of this thesis forked the project’s master branch and will develop the solutions on the respective fork. The listing 4.1 presents the address of multicore computing project in the GitHub.

1 https://github.com/maghsoud/IncludeOS

Listing 4.1: Multicore computing for IncludeOS project in GitHub

The development system requires to install git package and accordingly clone the IncludeOS project. Then the IncludeOS operating system should be built with existing bash script IncludeOS/install.sh. This installation will install all the necessary libraries and tools required by IncludeOS.
After installation, the existing system is ready as an Unikernel operating system. While this thesis follows the modular structure in mind, thus the solution development will be as a service for the existing system. The existing system has a service file which can be linked the operating system in the IncludeOS/seed/service.cpp location which add development to the operating system as a module. Hence, all the C/C++ program code will reside in the service.cpp file as a service for IncludeOS operating system. The following section will implement proposed solutions in this file along with other auxiliary files.

4.2 Development

Once the existing system is built and ready, the proposed solutions should be developed through program codes in C/C++ and Assembly languages. This section will build the functions in order to gather processor’s information, enumerate processor topology and programme APIC to send inter-processor interrupts to application processors. Here, the data types and core program codes will be discussed and the complete program codes will be enclosed as an appendix to this thesis.

4.2.1 Collecting Processor Basic Information

The software should look at different tables in order to get necessary information about the application processors (APs). According to Intel Multi-Processor Specification [46], the MP configuration table may contain information about the APs, thus software should examine the MP configuration table in order to find useful information.

4.2.1.1 MP Configuration Table

The first step in the multicore system in order to collect information is to check whether the system is multicore compliant. According to Intel MultiProcessor Specification [46], the MP configuration table would contain information about a multiprocessor system. Indeed, the existence of the MP configuration table indicates that the system is multicore compliant system and operating system can implement further actions in order to enable multicore computing on the system.

The operating system should access to the MP configuration table through the MP floating pointer. Indeed, The MP floating pointer is a gateway to the MP configuration table by containing a pointer to it. So, software should find the MP floating pointer in the first one megabyte of the memory. According to the Intel MultiProcessor Specification, software should investigate the first kilobyte of the EBDA or last kilobyte of the system base memory or BIOS ROM address space between 0F0000h to 0FFFFFFh in order
to find the MP floating pointer signature. The MP floating pointer contains "_MP_" ASCII string, so the software should investigate the specified memory for this ASCII string.

The first step for software is to find the bottom of the extended BIOS data area (EBDA) in the memory to inspect it for specified ASCII string. While the BDA and EBDA are partially standardized, thus it would be difficult to rely on their value. According to OSDEV.org 2014, the word value of 4-bit left shift to the 040Eh in the BDA would point to the bottom of EBDA. The listing 4.2 presents the C++ program code in order to retrieve the EBDA address.

```
1 *((uint16_t*)0x040E) << 4
```

Listing 4.2: Program code to retrieve the address of EBDA

The return value is 9FC00h while there is no MP floating pointer signature in the first kilobyte of this area. Thus, a reasonable approach would be traversing a memory area between the last kilobyte of the base memory (99CF0h) to the end of BIOS ROM address space (0FFFFFh) to find the MP floating pointer signature. This can be done with a simple while loop.

Afterward, the MP floating pointer structure should be stored in the memory for further usage. Thus, the struct data type could store the pointer into the memory for further usage. The structure can only hold the necessary values and can skip some of the entries. The listing 4.3 presents the data structure in C++ which contains the necessary entries of MP floating pointer structure.

```
1 typedef struct MPFloatingpoint {
2     uint32_t signature;
3     uint32_t MPtable_addr;
4     uint8_t Length;
5     uint8_t Version;
6     uint8_t FByte[5];
7 }MPFloatingpoint;
```

Listing 4.3: MP floating pointer data structure

In this structure, only five entries considered holding the necessary information about MP floating pointer. Software should check the value of the configuration table 1 which stored in the FByte[0] in order to check that this entry is cleared, thus the MP configuration table exist. The feature byte 2 is stored in the FByte[1] and the bit-7 is the IMCR flag then it indicates the type of interrupt mode in the system. Whenever bit-7 of feature byte 2 is set, then the PIC interrupts mode is implemented otherwise the virtual wire interrupt mode is implemented. Finally, the MPtable_addr data type indicates the address of the MP configuration table and software use this
value to find the table. Other entries in the MP floating pointer structure are informative so software may skip them. The figure 4.1 illustrates the values of MP floating pointer table entries on the system emulated by Bochs.

![Figure 4.1: MP floating pointer table Results](image)

The software uses the MP configuration table in order to get information about the multicore system. The table structure has 12 entries and software can use following struct data type in order to utilize them. This table begins also with a signature and it recognized by ASCII string “PCMP”. Followed by signature the BaseLength entry indicates the length of the base table which in this scenario, the base table length is 208 bytes. The Version entry indicates the version number of the MP configuration table and here this entry contains 4 which considered as version 1.4. The OEM entry identifies the creator of the table while the system emulated by the bochs, thus the OEM string returns BOCH. The Product_ID entry identifies the processor product family while the system is emulated by the Bochs, thus the value is SCPU0.1 which is the default value. The entry_count stores the number of entries in the variable portion of the base table that identifies the number of table entries. These table entries consist of a processor, bus, I/O APIC, I/O Interrupt assignment and local interrupt assignment tables and the other are reserved. Indeed, these tables build by BIOS but operating system could update them. For example, the processor entry table contains the BSP information and the operating system can add AP’s information. The processor entry consists of local APIC ID, a local APIC version, CPU enable and BSP flags, CPU signature, and CPU feature flags. The entry tables start at the end of the MP configuration table. The software can use the entry_count and the BaseLength values to calculate the address of the entry tables. The listing 4.4 presents the necessary entries of the MP configuration table data structure in C++.

The figure 4.2 illustrates the values of the MP configuration table entries on the system that is emulated by Bochs. While this table is built by BIOS, thus the information of the APs does not exist in the MP configuration table and operating system should update this table. However, this table will not help the software to get information about the application processors.
typedef struct mp_conf_table {
    uint32_t signature;
    uint16_t BaseLength;
    uint8_t Version;
    std::string OEM;
    std::string Product_ID;
    uint16_t OEMtable_size;
    uint32_t OEMtable_Point;
    uint16_t entry_count;
    uint32_t LAPIC_address;
    uint16_t EXTtable_length;
} mp_conf_table;

Listing 4.4: MP configuration table data structure

Therefore, the next step will be an examination of the ACPI specification in order to find useful information about application processors.

4.2.1.2 ACPI Specification

The ACPI specification tables contain information about the power management. This table is created by the BIOS during the system initialization and operating system may use information to configure performance and power consumption. The ACPI specification consists of different tables and operating system may use these tables in order to get hardware information. The ACPI tables are accessible through the main table called a root system description pointer (RSDP). This table is like a gateway to the ACPI specification chains of tables, thus software should find this table in order to access the other ACPI tables.

According to the ACPI Specification [41], the operation system should look for RSDP table in the first kilobyte of the extended BIOS data area (EBDA) otherwise, it should look at the BIOS memory address space between 0E0000h to 0FFFFFh. As far as the EBDA is partially standardized, thus it may not contain the RSDP table. While the RSDP table contains an ASCII string as a signature, thus the operating system can search the memory in order to find it. The operating system should search the low memory address space between 9FC00h to 0FFFFFh in order to find the ASCII string "RSDP" signature. The RSDP table structure can be as follows and only contain necessary entries. The listing 4.5 presents the necessary entries of the RSDP data structure in C++.

The RSDP table contains 8 entries and software can use the following structure in order to store their values. The signature entry must contain "RSD PTR" ASCII string which identifies the RSDP table. Following the signature, the OEM_ID value identifies the OEM. While the system emulated by Bochs, thus the OEM_ID contains the "BOCH" ASCII string. The Rev value identifies the revision number of the table and Bochs clears
If the Rev value is zero, then the revision number of the table is 1 otherwise revision number is 2. The RSDT_addr and XSDT_addr contain the physical address of RSDT and XSDT respectively. Finally, the Length entry should contain the size of the table, including a header in bytes but it is 0 while the system emulated by Bochs. The figure 4.3 illustrates the RSDP returned value while system emulated by Bochs.

The software uses the RSDT_addr value in order to find the root system description table (RSDT). The RSDT contains the system description header and an array of 32-bit physical address pointers to other description headers. Hence, the RSDT is a gateway to access other ACPI tables. The listing 4.6 presents the necessary entries of RSDT data structure in C++ language.

The RSDT structure consists of 9 header entries which are common in

Figure 4.2: MP configuration table Results
typedef struct {
    std::string signature;
    std::string OEM_ID;
    uint8_t Rev;
    uint32_t RSDT_addr;
    uint32_t Length;
    uint64_t XSDT_addr;
} RSDP;

Listing 4.5: RSDP data structure

Figure 4.3: Root system description pointer (RSDP) table results

all other tables. The software may store some of these entries in order to retrieve required information. The signature should contain "RSDT" ASCII string to identify the RSDT. The OEMID entry identifies the OEM while the system emulated by Bochs, thus its value is a "BOCH" ASCII string. The OEM_TID identifies the OEM table manufacturer ID ASCII string while the system emulated by Bochs. The Length entry specifies the size of RSDT in bytes which is 52 bytes while system emulated by Bochs. The Entry
typedef struct {
    uint32_t signature;
    uint32_t Length;
    std::string OEMID;
    std::string OEM_TID;
    uint32_t Entry;
} RSDT_table;

Listing 4.6: RSDT data structure

contains an n-4 bytes array of physical addresses point to another table. The length of the entry array should be calculated in regard to the value of Length. For example, the value of Length entry is 52 and the table header size in the system emulated by Bochs is 36, thus the entry size is 16 bytes. While each entry is 4-bytes long, thus the entry array contains 4 physical addresses. In the test system, the entry array contains the address of Fixed ACPI Description Table (FADT), Secondary System Description Table (SSTD), Multiple APIC Description Table (MADT) and IA-PC High Precision Event Timer Table (HPET). The figure 4.4 illustrates the RSDT entry’s value on the system emulated by Bochs.

In regard to figure 4.4, the RSDT contains only four pointers to other tables. This illustrates that the other ACPI tables are not available in the system emulated by Bochs. Since the multicore system can be initialized by advanced programmable interrupt controller (APIC) then the MADT will be an interesting table for APIC related information. However, the software can use other methods to gather information about multicore architecture. The main information source in the multicore system would be model specific registers (MSRs).

4.2.1.3 Model Specific Registers (MSRs)

Most of Intel IA-32 and 64 processors contain the model specific registers (MSRs). While the MSRs are readable and writable, thus the operating system can use them to configure the system. There are two groups of MSRs including architectural and non-architectural. The non-architectural MSRs are not guaranteed to be supported in the future.

The software can retrieve the multicore information through reading the MSRs. The local APIC plays an important role in the multicore initialization and software must collect information about it. The MP configuration and the ACPI MADT tables contain APIC information. In order to configure APIC in processors, the software requires the base address of the APIC registers in the system. The APIC base address can be retrieved from MP configuration and ACPI MADT tables through finding the tables in the low memory area. Otherwise, software can retrieve the APIC base address through IA32_APIC_BASE MSR. This 64-bit memory
Figure 4.4: Root system description table (RSDT) results

mapped register is accessible through RDMSR instruction.

The software should use the CPUID instruction (CPUID.EAX=01h:EDX[5]=1) in order to determine whether MSRs are supported in the system. Thus, it can use the RDMSR or WRMSR in order to read and write MSRs. The RDMSR reads the contents of 64-bit MSR, which MSR address specified in the ECX register into EDX:EAX registers. The EDX register will contain high-order 32 bits of the MSR and the EAX register will contain the low-
order 32 bits of the MSR.

The software can use RDMSR to read the IA32_APIC_BASE MSR in order to read the MSRs. The listing 4.18 presents the in-line assembly code in order to read IA32_APIC_BASE MSR by RDMSR instruction.

```
asm volatile("rdmsr": "=a" (EAX),"=d" (EDX) : "c" (0x1B));
```

Listing 4.7: Read the IA32_APIC_BASE MSR

The software can read the IA32_APIC_BASE MSR bit EAX[11] to determine whether the local APIC on the logical processor is enabled. Also, software can disable local APIC on a specific logical processor by clearing the EAX[11] then a reset will enable the local APIC again. The software should extract the local APIC base address through EDX[3:0]:EAX[31:12] with RDMSR instruction. The figure 4.5 illustrates the IA32_APIC_BASE MSR value in the bootstrap processor (BSP) of a system emulated by Bochs.

Figure 4.5: IA32_APIC_BASE MSR value on the BSP

The software retrieves information from the MSRs with RDMSR and will manipulate MSRs through WRMSR. But MSRs are architectural and software should use the CPUID instruction to determine the MSRs are exist. On the other hand, the CPUID instruction returns useful information about the processor through general purpose registers.

### 4.2.1.4 CPUID Instruction

The CPUID instruction provides information about the processor that it is running on. It identifies the features that the processor supports and return information about logical processors in a package. Hence, the CPUID is a handy instruction that returns much information to the software.

The software can use the EFLAGS register to determine whether the CPUID instruction is supported. If the software can change the value of bit-21 in the EFLAGS then CPUID instruction is supported. The EFLAGS can be stored in the stack by PUSHFD instruction and then load to general
purpose register by POP instruction. Thus, the software writes bit-21 of the
EFLAGS by storing the value on the stack through PUSH instruction and
load the value into EFLAGS register by POPFD instruction. The software
can implement the operation by the Assembly instructions. The listing 4.8
presents a program code in inline Assembly in order to determine whether
the CPUID instruction is supported.

```assembly
asm volatile("pushfd
" "pop %edx
" "movl %edx,%0
" "mov %edx,%eax
" "xor $0x200000, %eax
" "push %eax
" "popfd
" "pushfd
" "pop %eax
" "movl %eax, %1
" "push %edx
" "popfd
"
:"=m" (flagrd), "=m" (flagwr)
);
if (!(flagrd ^ flagwr))
return 0; // CPUID is NOT supported
return 1; // The CPUID instruction is supported
```

Listing 4.8: Determine whether CPUID instruction is supported

The CPUID instruction takes no parameter in Assembly language and
it implicitly uses EAX and ECX registers to determine the category of
the returned information which is called a leaf. The CPUID instruction
returns processors information in the general purpose EAX, EBX, ECX and
EDX registers. The software should use Assembly code in order to get
processor information through CPUID instruction. The listing 4.9 presents
the CPUID instruction in inline Assembly so that it can be used during the
development.

```assembly
asm ("cpuid": "=a" (reg[0]), "=b" (reg[1]), "=c" (reg[2]), "=d" (reg[3])
: "a" (eax), "c" (ecx)
);
```

Listing 4.9: CPUID instruction

In the inline assembly the a, b, c and d reflects the EAX, EBX, ECX and
EDX registers respectively. It is notable that first colon after instruction
determines the output that indicated by an equal sign in order to be
writable and second colon after instruction determines input registers.
The CPUID has a different number of leaves and each leaf returns specific information about the processor and the number of leaves supported is model specific. Hence, the software should determine that how many leaves are supported by the processor, otherwise the processor clears registers if the input value of EAX register is more than supported leaves. However, CPUID instruction returns the maximum number of supported leaves into the EAX register while the input value of EAX is zero (CPUID.EAX=0:EAX[31:0]).

It is notable while the virtual machine is created by QEMU then emulated processor by QEMU supports only 4 levels even though the host processor supports more leaves. Hence, QEMU should emulate the host processor by issuing the "-cpu host" option in order to virtual machine get the benefits of a full feature hardware processor.

There are tow main categories of leaves which are basic and extended leaves. The basic leaves explicitly depend on the input value of EAX while the extended leaves implicitly depend on ECX register. However, the processor topology enumeration is on the extended leaves and the return value depends on EAX and ECX input.

The software can use the CPUID instruction in order to get information about the processor that it is running on and in some cases information about the whole package. Hence, software can identify processor and store multicore information in the memory for further use such as processor vendorID, the number of logical processors and so on.

### 4.2.1.5 Vendor Identification String

The first step to gather processor information is to identify vendor identification string (vendor ID). Each processor has a specific vendor ID so that software can determine how to treat the processor. Thus, software can use CPUID leaf zero to retrieve the processor vendor ID. The CPUID leaf zero returns the processor vendor ID as an ASCII string in EBX, EDX, and ECX register. Each of the registers contains 4 ASCII characters of the vendor ID, so the vendor ID is a 12-character ASCII string. For example, the vendor ID of the Intel processors is "GenuineIntel". The listing 4.10 presents the Intel vendor identification string returned by CPUID instruction in the EBX, ECX, and EDX general purpose registers.

```
1  EBX = "Genu"
2  ECX = "ntel"
3  EDX = "ineI"
```

Listing 4.10: Intel vendor ID

The table illustrates a complete list of processor vendor ID. If the CPUID leaf zero does not return the valid vendor ID, then higher CPUID leaves are
not reliable. The listing 4.11 presents the program code in order to retrieve the processor identification string through the CPUID instruction. Hence, the software may use this inline Assembly code in order to retrieve the processor vendor ID in the system.

```c
uint32_t VID[3] = {0,0,0};
asm ("cpuid" : "=b" (VID[0]), "=c" (VID[2]), "=d" (VID[1]) : "a" (0));
std::string IDD = (char*) &VID[0];
IDD = IDD + ((char*) &VID[1]) + ((char*) &VID[2]);
IDD.erase(12);
```

Listing 4.11: Program code in order to retrieve Vendor identification string

### 4.2.1.6 Hyper-Threading Technology

The Hyper-Threading Technology improves processor performance while introduces logical processors inside the processor package to execute concurrent code streams. The logical processors share processor cores resources while each logical processor has its own sets of registers, local APIC and MSRs. The Hyper-Threading Technology introduces a multicore environment within a processor package. Thus, software must determine whether the processor supports the Hyper-Threading Technology in order to implement multicore computing within a processor package.

The software can use CPUID instruction leaf 1 in order to identify processor supported features. The CPUID instruction leaf 1 returns processor feature information in the ECX and EDX registers. The figures 4.6 and 4.7 illustrates processor supported features on a QEMU virtual processor which emulated the host Intel® Core™ i7-3632QM processor on an Ubuntu operating system. The listing 4.12 presents the program code in order to determine whether the Hyper-Threading Technology exist.

### 4.2.2 Advanced Programmable Interrupt Controller

The advanced programmable interrupt controller (APIC) is a unit to control interrupt and consists of local APIC and I/O APIC. There is one I/O APIC in the system bus while each logical processor connected to system bus has its own local APIC unit. This architecture allows logical processors to build a multicore environment that conforms with MP specification.

The software can use IA32_APIC_BASE MSR bit 11 flag in EAX[11] to determine whether the local APIC is enabled. Also, the software may use the CPUID instruction leaf 1 to determine the existence of APIC. The return
uint32_t EDX=0;

if (( CPUIDsupported() == 1) && (VendorID()=="GenuineIntel"))
{
    asm ("cpuid" : "=d" (EDX)
         : "a" (1)
         );
    if( EDX & HW_HTT_BIT)
        return 1;
    else
        return 0;
}
else
    return 0;

Listing 4.12: Program code to determine the Hyper-Threading Technology support

Figure 4.6: The Intel® Core™ i7-3632QM processor supported feature on ECX inside the QEMU virtual machine

value of EDX[9] indicates whether the processor support on-chip APIC in regard to default memory mapped registers in the physical address range FFFE0000h to FFFE0FFFh. The return value of ECX[21] indicates whether the processor supports x2APIC. Although the APIC holds 8-bit APIC ID the x2APIC holds 32-bit APIC ID and can address near 4294967280 processors in the multiprocessor environment.

In order to access logical processors in a system, local APIC units
Figure 4.7: The Intel® Core™ i7-3632QM processor supported feature on EDX inside the QEMU virtual machine

require unique identification called APIC ID. The APIC ID complies with processor topology enumeration and can address all logical processors in the package. The software can retrieve logical processor, initial APIC ID through CPUID instruction leaf 1. The CPUID leaf 1 returns the initial APIC ID of the logical processor, which the CPUID runs on into EBX[31:24] register.

The APIC ID assignment in a system is based on the system topology and consists of simultaneous multithreading (SMT_ID), cores ID and packages ID. Therefore, software should determine the number of logical processors and cores in the system in order to make topology enumeration.

### 4.2.3 Logical Processor Enumeration

In the virtualized environment, hypervisor allocates logical processors to the virtual machines as virtual processors (VCPUs), so software should enumerate the number of available logical processor in the system.

The software can use the CPUID leaf 1 in order to enumerate the maximum number of addressable IDs for the logical processor introduced to the virtual machine. The CPUID leaf 1 returns the maximum number of addressable logical processors in the EBX[23:16] registers. Thus, the nearest power of 2 integer that is equal or greater than the EBX[23:16] determines the width of the SMT_ID in the initial APIC ID. The table 3.2 illustrates the
hypothetical system with a dual processor, which each processor provide three cores and each core supports Intel Hyper-Threading Technology. In this hypothetical example, the SMT_ID width is 1-bit.

The software should identify the mask width of the SMT_ID so that processor can use this information for later performance tuning consideration. In order to achieve the optimal performance, software should send tasks to first logical processors in each core thereafter distributes tasks to the second logical processors in each core. The listing 4.13 presents the program code such that it determine the mask width of an input value. Hence, the software may use this inline Assembly code in order to find a mask width of the SMT_ID and Core_ID.

```
1 unsigned int mask_width = 0, cnt = count;
2 asm ( "mov %1, %eax \n"
3 "mov $0, %ecx \n"
4 "mov %ecx, %0 \n"
5 "dec %eax \n"
6 "bsr %ax, %cx \n"
7 "jz next%= \n"
8 "inc %cx \n"
9 "mov %ecx, %0 \n"
10 "next%= : "
11 "mov %0, %eax"
12 : "=r" (mask_width)
13 : "r" (cnt)
14 );
15 return mask_width;
```

Listing 4.13: Determine mask width of SMT_ID and Core_ID

The program code uses the BSR Assembly instruction in order to scan sets bit in the reverse mode. Therefore, BSR instruction scans a register from high order to low order and return the position of first bit that is set.

4.2.4 Processor Core Enumeration

The logical processor enumeration through CPUID instruction returns the maximum logical processors in the virtual machine. Thus, software should enumerate the number of cores in order to illustrate the processor topology hierarchy.

The software can use CPUID instruction leaf 4 in order to enumerate the maximum number of addressable core IDs in the physical package. The CPUID leaf 4 returns the maximum number of core IDs in EAX[31:26] register. If the processor does not support the CPUID leaf 4, then the software should assume that this is a single core processor. The listing 4.14 presents a program code in which the software can enumerate maximum addressable processor cores in a package.
Listing 4.14: Determine number of processor’s cores

```c
unsigned int EAX=0;
if (HTT_supported())
{
    asm ("cpuid": "a" (EAX)
         : "a" (4), "c" (0)
    );
    return (((EAX & 0xFC000000) >> 26) +1);
}
else
    return (1); // the processor must be a single-core
```

4.2.5 Processor Package Enumeration

In a system may exist two or more physical processor packages. The package ID sub-field in the APIC ID identifies different physical processors within a cluster. While the hardware does not belong to a multicluster environment, thus the remaining field in the APIC ID will be assigned to package ID. After that, the software calculated the SMT_ID and Core_ID bit width, the remaining bit-field of APIC ID will be the Package ID.

4.2.6 Processor Topology Hierarchy Mapping

According to *Intel® 64 and IA-32 Architectures Software Developer’s Manual: Instruction Set Reference, A-M* [76], the CPUID instruction leaf 0Bh identifies the processor topology hierarchy. The leaf 0Bh is dependent on the input value of the ECX which determines the level type of the topology. The return value of the ECX[15:8] returns the level type of the processor topology hierarchy. Software should check the return value in ECX[15:8] whether the CPUID returns a valid topology hierarchy. If the ECX[15:8] output value is zero then the output value of EBX register is also invalid.

In the virtual machine which emulated by QEMU, all input values of ECX return invalid level types in ECX[15:8]. Thus, the leaf 0Bh is not useful for retrieving the processor topology hierarchy in the virtualized system by QEMU. The listing [4.15] presents a program code in order to map the processor topology. However, the software may use this program code in order to extract processor hierarchical topology through initial APIC ID.

It is notable that, if the system uses x2APIC then the processor topology can be retrieved from CPUID leaf 0Bh.

4.2.7 Multicore Initialization

Following the system power-up or reset, each processor connected to system bus performs build-in self-test(BIST) process. During the BIST
uint32_t APICID = 0, MaxLP = 0, Maxc = 0;
unsigned int LPmask = 0, coremask = 0, SMT_ID = 0, Core_ID = 0, Package_ID = 0;
APICID = InitialAPICID();
MaxLP = MaxLogicalProcessor();
coremask = find_maskwidth(Maxc);
LPmask = find_maskwidth((MaxLP / Maxc));
SMT_ID = (APICID & ((1 << LPmask) - 1));
Core_ID = ((APICID >> LPmask) & ((1 << coremask) - 1));
Package_ID = ((APICID >> (LPmask + coremask)))

Listing 4.15: Processor Topology Hierarchy Mapping

process, one of the logical processors that connected to system bus selected as a bootstrap processor (BSP) and the others designated as application processors (APs). While the APs have been avoided to execute the operating system, thus BSP is responsible for executing the operating system and start-up APs. The APs stay in the halting mode while they are interrupt disabled. The APs respond only to INIT inter-processor interrupt (IPI) and STARTUP IPI in order to be initialized.

While the APs become initialized with IPI and the local APIC is responsible for controlling the interrupts, thus APIC is the central unit in the MP initialization process. The software can configure APIC through APIC memory mapped registers. The software should follow initialization protocol in order to make the APs ready for task execution. The initialization protocol would be as follows.

- BSP defines a counter to count the initialized APs.
- BSP sends a directed or broadcast initialization IPI to APs.
- BSP sends a directed or broadcast STARTUP IPI which contains a vector to address a memory location that self-configuration code resides.

It is very important that between INIT IPI and SIPI must be a programming delay in order to APs has enough time to initialize their registers. Otherwise, the STARTUP IPI can not lead APs to execute self-configuration code. According to Intel MultiProcessor specification [46], the operating system should implement following initialization order to wake up APs:

- BSP sends APs an INIT IPI
- BSP performs 10 mSec delay
- If the APIC is not 82489DX then BSP sends STARTUP IPI (SIPI)
- BSP performs 200 μSec delay
• BSP sends again the same SIPI
• BSP performs 200 µSec delay. The delay should be enough then all
  APs could initialize themselves and increment counter

If the APIC is an 82489DX discrete unit, then it will ignore the SIPI so
software should determine the APIC through the memory map local APIC
version register[7:0]. If local APIC version register[7:0] is less than 10h then
it is discrete 82489DX APIC and operating system should use BIOS in order
to wake-up the APs.

The INIT and SIPI are open-ended commands and operating system should
determine whether they executed correctly. The Operating system should
guarantee INIT and SIPI successful dispatching through the delivery
status bit in the interrupt command register (ICR) or APIC error status
register[7:0].

The software must use the ICR in order to trigger INIT and SIPI toward
APs. The ICR is a two 32-bit registers known as ICR-high and ICR-low
which reside at offset 310 and 300 from the APIC base address.

4.2.7.1 Interrupt Command Register

The interrupt command register (ICR) is a 64-bit local APIC memory
mapped readable and writable register which allows software to issue IPI
to all processors connected to the system bus. The IPI will trigger if the
ICR-low 32-bit register has been written. So the software should first write
high 32-bit ICR register, then writing to the low 32-bit ICR will trigger the
IPI.

The high 32-bit ICR contains only the APIC ID of the destination processor
and software will use it in order to trigger directed IPI to another AP. The
high order 32-bit ICR resides in 310h offset from the APIC base address
while destination field is 8-bit wide in an APIC mode and 32-bit wide
in x2APIC mode. If the destination field is filled with FFh value, then it
operates as broadcast IPI.

The low order 32-bit ICR resides in 300h offset from the APIC base address
which contains the configuration flags and writing in this 32-bit will
trigger IPI. This 32-bit register contains vector[7:0], delivery mode[10:8],
destination mode[11], delivery status [12], level[14], trigger mode[15] and
destination shorthand[19:18] fields. The 8-bit vector field specifies the
vectorized address of the self-configuration code that targets APs jump
there after receiving the SIPI. This field is ignored during the INIT IPI and
is only meaningful during the SIPI. The vector field contains VV vectorized
address of the self-configuration code in the first megabyte of the memory
in the form of 000VV000h format. Therefore, the vectorized address must
be in a 4-kilobyte memory boundary. For example, if the self-configuration
code resides on the 80000h memory address the vector field should contain
80h during the SIPI in order to lead APs to execute self-configuration code in the 80000h memory address.

The delivery mode field determines the type of outgoing IPIs. This 3-bit field can address 6 interrupt mode, including fixed, lowest priority, SMI, NMI, INIT and STARTUP IPIs. The 101 binary value of delivery mode determines INIT and 110 binary value of delivery mode determines STARTUP IPI.

The destination mode determines the mode of addressing. This flag determines whether physical or logical addressing mode should be used in order to address target local APIC IDs. If this value is cleared, then local APIC uses physical APIC IDs in order to address APs. Otherwise, the local APIC uses logical addressing in conjunction with local destination register (LDR) and destination format register (DFR). While the APIC IDs is known, thus it is recommended that the software uses physical addressing so this flag should be cleared.

The delivery status flag holds the status and it is used to monitor ICR status. This flag should be used in order to guarantee successful IPI dispatching. The software should check this flag after sending IPI to determine whether the IPI is dispatched successfully. It is notable that the dispatching process will take approximately 20µSeconds. Hence, the flag would be set if the sending is pending.

The level flag determines whether the APIC should activate the IPI in a de-assert mode or assert mode. The de-assert mode activates IPI with a high signal while assert mode activates IPI with the low state signal. In older processors this flag should be cleared for INIT IPI and set for all other delivery modes. But, on a Pentium 4, Intel Xeon and newer processor families this bit must be always set.

The trigger mode flag determines whether the IPI should trigger with an edge of signal or level. While IPI should trigger as soon as possible, thus this flag should be cleared in order to trigger the IPI with a first edge of the signal.

The destination shorthand determines the destination of the IPI. The 00b binary value of destination shorthand indicates that the local APIC must read the destination APIC ID from the high order 32-bit ICR[31:21]. The 01b binary value of destination shorthand indicates that the destination of IPI is the same local APIC that triggers it. The 10b binary value indicates that local APIC should broadcast the IPI on the system bus and all logical processors including the sender are as the destination. The 11b binary value indicates that local APIC should broadcast the IPI in the system bus and the sender must not react to the IPI.

In the ICR, all other fields are reserved and must be cleared. It is important that ICR requires two writes to trigger directed IPI to other APs. In all other destination shorthand, one writes to ICR low double word register will suffice to trigger the interrupt.
4.2.7.2 INIT IPI

The software can trigger an INIT IPI to APs though setting the low ICR register[10:8] to a binary value of 101b. The INIT IPI will initialize target AP's registers and put the processor in real-addressing mode. Hence, vectorized address in the SIPI should point to a low memory address. The figure illustrates the low ICR in order to trigger INIT IPI to all APs.

In the figure, the delivery mode field set to 101b to indicate the INIT IPI. The destination mode and delivery mode fields cleared. The level flag set to assert mode while the trigger mode flag cleared. The destination shorthand set to 11b in order to send an IPI to all excluding the sender. In regard to broadcasting the IPI in the system bus, the value in the destination field in high order ICR will be ignored. Hence, the software does not require to write to high order ICR and only configure the low order ICR to send an INIT to all APs. The software can use the following code to write low ICR. The listing presents program code in order to broadcast INIT IPI such that the APIC_BASE_addr variable indicates the APIC base address and the ICR_low macro indicates the offset of the ICR low double word register.

```
1 *((uint32_t *)(APIC_BASE_addr | ICR_low))=0x000C4500;
```

Listing 4.16: ICR low write operation to issue INIT IPI

After INIT, the APs will wake up from the halt mode and initialized their registers in real-addressing mode. Hence, there should be an enough programming delay in order to APs initialized their internal components

Figure 4.8: The ICR value while sending the INIT IPI to all APs

In the figure, the delivery mode field set to 101b to indicate the INIT IPI. The destination mode and delivery mode fields cleared. The level flag set to assert mode while the trigger mode flag cleared. The destination shorthand set to 11b in order to send an IPI to all excluding the sender. In regard to broadcasting the IPI in the system bus, the value in the destination field in high order ICR will be ignored. Hence, the software does not require to write to high order ICR and only configure the low order ICR to send an INIT to all APs. The software can use the following code to write low ICR. The listing presents program code in order to broadcast INIT IPI such that the APIC_BASE_addr variable indicates the APIC base address and the ICR_low macro indicates the offset of the ICR low double word register.

```
1 *((uint32_t *)(APIC_BASE_addr | ICR_low))=0x000C4500;
```

Listing 4.16: ICR low write operation to issue INIT IPI

After INIT, the APs will wake up from the halt mode and initialized their registers in real-addressing mode. Hence, there should be an enough programming delay in order to APs initialized their internal components
completely. If the delay is not enough, then the SIPI would not lead the APs to execute self-configuration code in the low memory.

The software can use programmable interval timer (PIT) to make delay after INIT IPI. The PIT consists of three channels with individual counter including timer interrupt channel, system use channel and CPU speaker channel. The channel 0 is connected to IRQ 0 and it generates interrupts at a fixed time. Channel 1 was used for refreshing the DRAM in earlier personal computers and it is not a common solution for making delay. The channel 2 is connected to PC speaker to generate beep at a given frequency. The channel 2 is the only channel where the operating system can control the gate input without enabling interrupts. This channel uses port 42h as the data port and 43h as command register. The oscillator in PIT running at 1.19318166 MHz and divided by a 3-bit divider field. The I/O ports are 16-bit wide registers and software read and write to I/O ports through 8-bit register such as AL and AH registers [77–79].

The software can use the channel 2 gate in order to make delay. If the gate input state changes, then the output of the timer 2 bit 5 will change consequently. The change in input gates takes some times in order to be visible at the output, and this generates enough programming delay. The listing 4.17 present the program code in inline assembly in order to generate programming delay for inter-processor interrupts through PIT timer 2.

```
asm(    "in $0x61, %al \n"      
      "and $0xFE, %al \n"      
      "out %al,$0x61 \n"      
      "or $1, %al \n"       
      "out %al, $0x61 \n"
      "delay: \n"
      "in $0x61,%al \n"     
      "and $0x20,%al \n"    
      "jz .delay " );
```

Listing 4.17: Programming Delay

In this delay function, the software reads the port 61h to AL register, then clear bit 0 AL[0] in order to change the gate input state and set the bit 0 AL[0] in order to change gate input state back. Afterward, software should read the port 61h in order to determine whether bit 5 is set. The state change between gate input and the output generates enough delay for APs to initialize themselves.

4.2.7.3 STARTUP IPI

The STARTUP IPI is an interrupt with the delivery mode set to 110b and trigger mode flag cleared. The SIPI should be triggered while the APs have
been initialized with integrated local APIC. The software must generate programming delay after INIT and then trigger SIPI with the vectorized address of self-configuration code. The startup vector is limited to 4-kilobyte page boundary in the low memory address while the vectors A0-BF can not be used by SIPI. While SIPI uses the VV as its vector value, thus it configures CS:IP to VV00:0000h at the target processor. This leads the target processors to jump immediately to the memory address at VV000h and execute the self-configuration code.

The software must use ICR in order to send SIPI to other processors. The SIPI follows the same ICR as INIT except the delivery mode should be 110b to indicate the STARTUP IPI and it contains VV vectorized address of self-configuration code. The software can use the following code in order to trigger SIPI to all APs connected to the system bus. The listing 4.18 presents the code such that the APIC_BASE_addr variable indicates the APIC base address and the ICR_low macro indicates the offset of the ICR low double word register.

```
*(((uint32_t *)(APIC_BASE_addr | ICR_low)))=0x000c4680;
```

Listing 4.18: ICR write operation in order to issue STARTUP IPI

Here, the self-configuration code is in the low memory address 80000h, so the vector value in the SIPI should be 80h. After SIPI, there should be a programming delay in order to APs execute the self-configuration code completely and increment the counter. The self-configuration code would push the target processor to protected mode in order to make it ready for task execution.

### 4.2.7.4 Self-Configuration Code

The self-configuration code enables the processor to jump to the protected mode. Indeed, it is likewise a bootloader so that it load specific tables and register of application processor with a particular value. It leads the application processors to get ready for task execution in the operating system.

During the execution of trampoline code all interrupts are disabled and segment registers filled with code segment value. Thus, each processor should increment the counter in order indicates its presence during the trampoline execution. While the SIPI is broadcasted among the APs connected to the system bus, thus it may generate conflict during the simultaneous writes to counter. According to Intel® 64 and IA-32 Architectures Software Developer’s Manual: System Programming Guide, Part 1, the trampoline code should implement a semaphore in order to deal with race condition while incrementing the counter. As the semaphore method requires a spinlock, then it would introduce new problems in the
virtualized operating system. The hypervisor may pause a virtual CPU while it holds a spin lock. As long as the spin lock acquired by paused virtual CPU, thus other virtual CPUs on the same guest that trying to acquire the lock, must wait until they paused virtual CPU executed again and release the lock. Since more virtual processors on the guest machine running in parallel the more virtual processors must wait while trying to acquire the lock. Hence, the spin lock mechanism will waste virtual CPUs processing time in the virtualized operating system.

In order to deal with the race condition, the software can use the LOCK instruction while APs try to increment the counter variable. According to *Intel Software Developer Manual* [76], the LOCK instruction turns the accompanying instruction into an atomic instruction. In the multicore environment, the LOCK signal asserted and ensures that the processor has exclusive use of any shared memory area. The software should consider that the LOCK instruction can be prepended only to the instructions where the destination operand is a memory operand.

During the trampoline code execution, APs load the address of global descriptor table (GDT) and interrupt descriptor table (IDT) into GDTR and IDTR respectively. The software can store these registers in the BSP and share the memory address of the GDTR and IDTR values to the APs so that they use the same tables. The GDT is specific to IA32 architecture and it contains entries in order to translate a memory segments address for the processor. Indeed, the GDT defines the characteristics of segmented memory, such as access privilege and base address. The 6-bytes memory location GDT Register (GDTR) holds the 32-bit base address and the 16-bit limit (lower 2-bytes of the 6-bytes GDTR) of the GDT. The same architecture exists for IDT so that the 6-bytes memory location IDT Register (IDTR) holds the 32-bit base address and the 16-bit limit of the IDT. The GDTR can be stored through SGDT instruction and will be loaded through LGDT instruction. The IDTR can be stored through SIDT instruction and will be loaded through LIDT instruction.

Afterward, the trampoline code must lead the APs to the protected mode. In the protected mode, the operating system operates at both page level and segment level. The protection mechanism restricts access to certain pages and segments based on four privilege levels for segments and two privilege level for pages. In the protected mode, each memory reference passes various memory protection checks and any violation results in an exception. The processor switches to protected mode by setting the PE flag in CR0 control register and enables the segment-protection mechanism. Thus, there is no control bit to turn off the protection mechanism so sending the INIT IPI leads the processor to start in the real-addressing mode again. The software can set the CR0[0] in order to switch to protected mode.

After switching to protected mode, the software requires the APs run in the kernel mode. In order to run in the kernel mode, software should use the GDT selectors. There are 5 selectors in the GDT including NULL selector, code segment selector for kernel mode, the data segment selector for kernel
mode, the code segment selector for user mode and data segment selector for user mode. These selectors are all 8-byte in size so the selector indices are as follows.

- 0x00 NULL descriptor
- 0x08 Kernel code segment
- 0x10 Kernel data segment
- 0x18 User code segment
- 0x20 User data segment

Hence, the software must fill the code segment and data segment with 0x08 and 0x10 values, respectively in order to force the APs to run in the kernel mode. In order to fill the code segment with 0x08, the software should implement a far jump to 0x08:IP address. Changing the data segments are straightforward, so it can be done by simple MOV instruction.

Afterward, the application processors require a memory location for stack in order to store temporary data. The stack definition is quite straightforward in Assembly so that it requires to fill the stack pointer register (ESP) with the base address of the stack. This creates a stack for application processor so that they can use it to store temporary data. The size of the stack is important and it should be large enough. If the stack size is smaller than the stored data then the stack overflow problem occurs. The stack overflow can cause the old data overwritten with newer data. This is because the stack definition only requires the base address and does not include the size in order to prevent the stack overflow. Since this project is under development then it assumed that the 4-kilobyte stack will be enough in order to hold the temporary data for each application processor.

The stack can be defined in the real addressing mode or in the protected mode. The self-configuration code defines the stack in the protected mode after that the application processors entered the kernel mode. Since the self-configuration code should be run simultaneously by multiple application processors thus the self-configuration code should partition the processor’s stack from each other. The solution is to create an array of stacks which are indexed with APIC ID such that each application processor access its stack by its APIC ID. Indeed, the application processors use their APIC ID in order to fill the dedicated address to the ESP.

Thereafter, the APs are ready to execute the instruction in the kernel mode and the operating system should compare the AP’s counter with the maximum logical processors in the system minus one in order to get assured that all APs are alive and ready to execute tasks. Hence, the operating system should distribute tasks to the APs and manage them through the IPIs.
4.3 Testing

This section discusses the procedure of building and testing of solution development. The testing requires that the all program code pieces build together into a special file, then check if the development is usable, stable, predictable and achieve the project desire. Hence, the building process is a main part of the testing phase. Since the research is in permanent iteration process thus the building process should be automated with build tools. As discussed in the previous chapter, the IncludeOS project uses Make build automation tool in order to automate the build process. Since this research develop program codes in different files therefore, the building process develops the existing system Makefile in order to follow the same schema and uses Make tool for build automation.

4.3.1 Build

The make automation tool uses a specification file called Makefile. It describes the relationship between the source, intermediate, and the executable file so that make tool can perform the minimum amount of work in order to update executable file. By default, the Make tool looks for a file called Makefile in the working directory and executes it. So each time that part of some source files changed, the Make performs all necessary compilation. It uses the Makefile and last modification time of files in order to decide which files should be updated [80].

A Makefile may consist of only rules which each rule has target, prerequisites and recipe. The listing 4.19 presents A simple Makefile rule in which can execute a particular task.

```plaintext
1 target file : prerequisites
2   recipe
3   recipe
4   ....
5   recipe
```

Listing 4.19: Makefile rule

The target is usually an object file which is generated by a program or can be an action like clean. A prerequisite can be one or more files that are used as input to create the target. A recipe is an action which should be carried out and must proceed with a tab character. If any of prerequisite files is newer than target then target is recompiled by executing the recipe. Each recipe passes to the shell and executes in its own subshell. The compilation process can be automated through a Makefile rule for each target file. This procedure generates again several object files that should link together and build an executable image file [81].
The LD tool combines a number of object and archive files, relocate their data and knit symbol references inside files. The LD tool is usually the last step in building an executable. It uses the general purpose BFD libraries in order to read, combine and write object files in many different formats like a.out or COFF. The LD produces an executable as results of linking the input files and libraries. The LD tool uses command-line options in order to direct the linking operation which can be specified on the command line or as a script. If the linker does not recognize the format of object file then it assumes that it is a linker script. Every link is controlled by a linker script in the LD tool. The main purpose of the linker script is to map the sections of input files to the executable and control the memory layout of the executable file. Indeed, the linker script labels the memory address of the sections of input files and can be used in order to track the input files code in the executable file.

As an example, this research developed a self-configuration Assembly code in a separate file thus it should be compiled and linked with other files in a building stage. The listing 4.20 presents the Makefile rules in order to create a new object file for the self-configuration code and link it to the operating system image file.

```makefile
trampoline.bin: trampoline.asm
    @echo "\n>> Assembling trampoline"
    nasm -f bin -o $@ $<

trampoline.o: trampoline.bin
    objcopy -I binary -O elf32 -i386 -B i386 $< $@

service: $(OBJS) $(LIBS)
    @echo "\n>> Linking service with OS"
    $(LD) $(LDOPTS) $(OS_PRE) $(OBJS) $(LIBS) $(OS_POST)
    -o $(SERVICE)
    @echo "\n>> Building image " $(SERVICE).img
```

Listing 4.20: Makefile rules for self-configuration code

The first rule compiles the trampoline.asm Assembly input file to the trampoline.bin binary target file by NASM Assembly compiler. In this rule, the $@ option represents the target file and the $< represents the prerequisites. The second rule, converts the trampoline.bin binary file to an elf object file by Objcopy tool so that the linker can link it to the IncludeOS image. Because of the nature of self-configuration code in trampoline.asm file it should be compiled to the binary and consequently binary file converted to an object file.

The last rule links all the object files and libraries and builds an executable file. The service treats as an action and the $(OBJS) $(LIBS) variables represents all object files and libraries respectively. In the recipe part the $(LD), $(LDOPTS), $(OS_PRE), $(OBJS), $(LIBS), $(OS_POST) and $(SERVICE) variables represents LD tool, LD options including linker
script, pointer to constructors, object files including trampoline.o, libraries, pointer to destructors and the executable file respectively. The @echo is a directive for print an informative message during the building process.

This build process will compile and link developed program codes to the existing system and facilitates the testing process.

4.3.2 Development Testing

Once the build process finished, the executable must be tested on a machine equipped with Intel processor. Since the research focused on the Intel processors family thus the development should be tested on the machine with Intel processor inside. Thus, the research used a machine with Intel® Core™ i7-3632QM CPU equipped with 4 physical core and Hyper-Threading Technology and 8GB of memory. The processor operates at 2.2 GHz base frequency with 6MB Intel Smart Cache.

The testing phase checks the executable output to determine whether the development results are usable, stable, predictable and achieve the project desire. In order to test these factors, the testing part can use standard output of the executable and the QEMU monitor console.

The standard output can be used to print the informative messages accordingly use them to test the usability, stability, predictability and achievement of project aim. The printed messages and values in the standard output, test all of the criteria. Another kind of testing is done by QEMU monitor console in order to investigate the memory and the processor registers.

The QEMU monitor console provides interaction with QEMU through specific commands. The monitor console enables the user to investigate memory layout, processors register, processors status and control various aspects of the operating system running on. The user can switch to the monitor console within the QEMU by holding down the Ctrl-a + c keys. While the user switched to the QEMU monitor console thus there are following specific commands in order to investigate processors and memory:

- info cpus - shows the states of the processors and the value of the program counter.
- cpu n - switch to processor n so that other commands return the information of processor n.
- info registers - shows the value of processors registers.
- xp /fmt addr - shows the state of the working memory of the operating system. the /fmt (/countformatsize) is format which tells the command how to format data. The count is number of data to be dumped. The format can be i (Assembly instruction), x (hex) and c (char). The size can be b (8 bits), h (16 bits), w (32 bits) or g (64 bits).
If the size is not mentioned on x86 then the default would be 32 bits. The addr is the starting address for dumping. For example the $xp/10i 0x80000 dump 10 assembly instruction at the 0x80000 address [83].

The developer can use the standard output and the QEMU monitor console for the testing phase. The QEMU monitor console provides debugging capability in addition to the testing part and helps to debug the development part.

When the testing phase finishes successfully and the developments satisfy the research aim thus the research move on to the final stage which is the evaluation.

4.4 Evaluation

The evaluation phase executes experiments in order to demonstrate the advantages and disadvantage of each solution and judge them accordingly. In the previous chapter, the efficiency evaluation methodology discussed and decided that the time measurement should be the only kind of experiment implemented in the evaluation phase. This is due that the other efficiency factors like memory and disk can be calculated by a simple mathematical equation.

This thesis has proposed to use the multicore computing IncludeOS in contrast to multiple single core IncludeOS. Hence, the memory, disk and processor factors as resources can determine cost of each solution. In each operating system, \( n \) represent the number of tasks and \( P, M \) and \( D \) represent the cost of the processor, memory and disk in the operating system respectively.

In the multiple single core IncludeOS, if there are more than one task and \( n \) represents the number of tasks then \( n \) instances of operating system should be created in order to service all the tasks at the same time. At the same time, each IncludeOS instances uses the processor, memory and disk units. Therefore, \( n \) simultaneous tasks require \( n \) processor, memory and disk units.

In the multicore IncludeOS, if there are more than one task and \( n \) represents the number of tasks then \( n \) processor should be added to the operating system in order to service all the tasks at the same time. Since each IncludeOS instances uses processor, memory and disk units therefore, this solution requires only \( n \) processors beside the same amount of memory and disk as single core IncludeOS requires.

In contrast to memory and disk factors, the execution time factor should be achieved through a series of experiments. The experiments will facilitate the efficiency evaluation of each solution.
4.4.1 Building Experiments

As mentioned in the previous chapter, the experiments have various participants which are the multicore IncludeOS, the multiple single core IncludeOS, bare metal Ubuntu 14.04 and the Ubuntu 14.04 installed in a virtual machine. In the experiments, the input task is computing the nth prime number as a processor intensive task. One of the simple prime number algorithms is Sieve of Eratosthenes and this algorithm can be used in order to calculate the nth prime number.

The experiments perform on a machine with two Intel(R) Xeon(R) CPU E5-2699 v3 processors that each has 18 cores with Hyper-Threading Technology so that the machine contains 72 logical processors. The processor base working frequency is 2.3 GHz. The machine has also 128 GB of memory and 149 GB disk space.

However, the execution time of the task should be long enough so that it become comparable. Hence, the computation of 500000th prime number which is 7,368,787 will take a couple of seconds so that it satisfies the experiment aim. The prime function gets the nth value and returns the nth prime number.

On the other hand, the IncludeOS can only interact with the outside through the network adapter so that tasks should be feed in through network. Since the TCP protocol is under development thus it is not stable protocol to do experiments. The IncludeOS owns a simple and reliable UDP server so that UDP protocol has been selected as communication means with operating system during the experiments. On the other side, the experiments require the UDP client in order to send and receive the requests to the IncludeOS operating system.

In order to communicate with UDP server of IncludeOS, a UDP client has developed. The UDP client gets request and the server IP address as two arguments. The request argument is the nth prime number which in this experiment is 500000 and the IP address of the IncludeOS.

In order to measure the execution time of a task, the GNU Time tool employed with a UDP client script. The elapsed real time is used to measure the execution time and round trip time as total time during the sending task and receiving the reply.

The host operating system assigns different logical processor to guest virtual machine so that two virtual machines may use two logical processors in one core. Since the logical processors in one core share each core’s resource thus assigning two virtual machines to one core will reduce the performance.

All these configurations tune the experiments to generate the precise and reliable output. These configurations should perform for each operating system in the experiment and they should be automated. Since, the experiments are implemented on the machine running Ubuntu 14.04
operating system, thus configurations can be automated through Bash scripts. The experimental process can be divided into three bash script which boots the IncludeOS operating system by QEMU processes, measure the execution time of experiment and finally kill the virtual machines.

The first bash script will run the QEMU process with the IncludeOS image. In the experiment with multiple numbers of IncludeOS, each QEMU process should assign a specific IP address to the IncludeOS virtual machine. Since IncludeOS has not DHCP service, thus there is a problem with assigning a unique IP address to each virtual machine. The IP address can be derived from the MAC address of the network adaptor. This is due to that each QEMU process can start with unique MAC address so that each IncludeOS operating system can get a unique IP address through MAC address. Hence, the Bash script will assign different MAC address to each QEMU process through for loop.

The second script measures the total time of transmission and execution of the task and writes the results to a file. In the multiple single core IncludeOS case, the script opens a simultaneous sockets to multiple IncludeOS instances through Pthread model. This means that client sends the concurrent request to the multiple IncludeOS instances and sums the result of computation accordingly. In the multicore IncludeOS case, the operating system itself perform summation and returns the total results to the client.

The last script kills the IncludeOS instances after the test. The script will kill all the process IDs that belong to QEMU and running the IncludeOS image.

These scripts automate each step of the experiment and make sure that the each experiment, implement with same configuration each time. Since the experiments should be repeated many times and decided in the previous chapter that the experiments should be repeated at least 50 times in order to show that the results are predictable and stable. Therefore, another script can iterate the other three scripts in order to fully automate the 50 experiment.

On the Ubuntu operating system the processor intensive task executed through the POSIX thread (Pthread) model in order to execute tasks in parallel. Therefore, the processor intensive task can be executed simultaneously on different cores so that the multicore computing simulated in the Ubuntu operating system. The experiments on the Ubuntu operating system use the same nth prime number program code as the multicore and multiple single core IncludeOS Unikernel operating system utilized. Since the Ubuntu bootup process takes time so that Ubuntu boots up beforehand the experiments perform.

Finally, results of the experiments will be considered as the main basis in order to evaluate the efficiency of each solution. The upcoming chapter will discuss the results and analysis them in order to evaluate the solutions.
4.5 Summary

This chapter developed the multicore computing for IncludeOS Unikernel operating system. Each development step followed by the testing process to determine whether the development is stable and usable. Finally, a batch of experiments implemented in order to evaluate the development.
Chapter 5

Analysis

This chapter composes finding and experiment results in order to analysis the development and findings. The data were collected from experiments and then processed in response to the problem statement in chapter 1 of this thesis. The fundamental aim of this thesis motivates to collect data and analysis the development with competing solutions. The research aim is to develop multicore computing for cloud-based Unikernel operating system while so that it can be compared with other solutions.

5.1 Multicore Unikernel Operating System

This research developed multicore computing for IncludeOS Unikernel operating system. The multicore computing solution developed based on multicore processor system as a set of homogeneous processors. The research goal is to demonstrate that a multicore operating system will be an efficient solution to increase the computing power. The research retrieved the processor’s basic information in order to determine whether the processor is a multicore. Since the research has concentrated to Intel processors family, thus the Intel MultiProcessor Specification was the main reference. According to Intel MultiProcessor Specification, the first step to determine whether the system is a multiprocessor compliant system is to determine whether the multiprocessor configuration table exists. This motivated the research to look for this table in the low memory.

According to Intel MultiProcessor Specification, the MP configuration table should be within the first kilobyte of EBDA, last kilobyte of system base memory or BIOS ROM address space. The existing system uses Bochs as BIOS emulator and during the development, it was found that the Bochs places the MP configuration table in the BIOS ROM address space and software should traverse memory between $0F0000h$ and $0FFFFFFh$ to find the MP floating pointer signature. Since the MP configuration table and its subordinate tables have signature thus it is straight forward to traverse the
low memory space in order to find MP floating pointer structure as root to other multiprocessor tables

After finding the MP floating pointer structure, it contains an address which points to MP configuration table. According to Intel MultiProcessor Specification, it should contain the multiprocessor information. But during development, it was found that the operating system should update this table with multicore information otherwise it contains the default entries. The MP configuration table contains the APIC register base address, table creator ID and a number of subordinate tables after this table. The figure 4.2 demonstrates that APIC base address is FEE00000h and entry count value demonstrates that there are subordinate tables after MP configuration table. But these information is only about the bootstrap processor and there is no extra information about application processors. Although the APIC register base address is valuable information in order to start multicore development, but there is no information about the APs. Indeed, the software can not find out information such as the number of application processors or APIC ID in the MP configuration table.

While the operating system starts to develop the multicore computing thus the MP configuration table would not be helpful that much. According to Intel MultiProcessor Specification this table should be updated by operating system otherwise it contains default values and its existence demonstrates that the system is multiprocessor compliant and accordingly system is multicore compliant. In regard to finding during development, this chapter will demonstrate that there are other ways in order to collect default information resides in the MP configuration table.

Although the MP configuration tables exist, there other tables which created by the BIOS in the system during the power up and can be utilized. The ACPI specification was developed in order to standardize the device configuration and power management. Since the ACPI evolved and encompass the multicore specification, thus it is one of the important sources of information for the operating system. The ACPI consists of a chain of tables that each contains specific device information. The ACPI specification consists of 25 description tables and 23 reserved description header for future use. The MADT table is the most relevant table for multicore development, but it should be accessed through the RSDP.

The RSDP in the ACPI specification is like MP floating pointer structure in the MP configuration table and resides in the low memory address space. According to ACPI Specification [11], the RSDP should reside in the first kilobyte of EBDA or BIOS ROM area in the low memory. Since the existing system used Bochs as BIOS emulator, thus Bochs has created the ACPI description tables in BIOS ROM area and the software should traverse memory space between 0F0000h and 0FFFFFFh in order to find RSDP. The ACPI tables like MP configuration table begins with ASCII signature, which enables software to distinguish specific table.

The RSDP contains information about table creator ID, RSDT address and
if the there are extended ACPI tables. The root system description table contains the creator ID and address of the other ACPI tables. The figure 4.4 demonstrated that the emulated system contains only 4 ACPI description tables which contain device information. The emulated system consists of FADT, SSDT, MADT and HPET so that the MADT tables is the most relevant table for multicore development.

The multiple APIC description table contains the table creator ID, APIC register base address and information about the system interrupts controllers. Since the system is emulated by Bochs so that the creator ID is Bochs. The APIC register base address is same that MP configuration table contains. This table has major differences with MP configuration table that it contains information about all interrupts controllers in the system. This means that MADT contains the information about local APIC, I/O APIC, interrupt source override, NMI interrupt source, local APIC NMI, local APIC address override, I/O SAPIC, local SAPIC, platform interrupt source and processor local x2APIC information.

The multicore development requires the local APIC as the main player in inter-processor communication. Thus, at this point the information about the local APIC suffice the development of multicore computing. Though the I/O APIC is responsible for external interrupts, but the local APIC is interrupt management unit for each logical processor so that it is a gateway for accessing the logical processors.

Both ACPI tables and MP configuration table contain information about bootstrap processor’s local APIC. The most important information about local APIC is the base address of local APIC register. Hence, this can be achieved by an alternative way such as MSRs and CPUID instruction.

The model specific registers are memory mapped registers in the processor, that facilitates the configuration. In a multicore processor, each logical processor contains MSRs in order to facilitate processor configuration. The IA32_APIC_BASE MSR contain information about the local APIC register base address and provide software to disable or enable the local APIC in the processor.

The IA32_APIC_BASE MSR contain necessary local APIC information which the MP configuration and ACPI MADT tables present. The software reads and writes to MSR through RDMSR and WRMSR instruction respectively. Hence, software can use IA32_APIC_BASE MSR instead of traversing memory for MP configuration table and MADT ACPI in order to retrieve the local APIC information.

The figure 4.5 represents the content of IA32_APIC_BASE MSR which contains the local APIC register base address, bootstrap flag, and the local APIC enable flag. If the processor provides the IA32_APIC_BASE MSR then there is no need to look for the MP and ACPI tables.

Since the MP and ACPI tables do not contain information about the number of processors available to the operating system, thus another mechanism
should be employed in order to find the maximum number of processors in the system. The CPUID instruction presents information about the processor and it is almost standardized among the different processor vendors. Hence, it is a reliable and valid source of information about the processors.

The CPUID instruction returns various information about the multicore system. The software can use the processor vendor identification string in order to determine the target processor. This helps the software to specify the development for different processor vendors and do not generate a fault.

The CPUID instruction demonstrates which feature processor provides to the operating system. The figure 4.6 and 4.7 illustrate the feature supported by the Intel® Core™ i7-3632QM processor returned in the ECX and EDX registers. This information will help the software to determine whether the processor is a multicore package. Indeed, the Hyper-Threading Technology feature demonstrates that processor is a multicore package otherwise it is a single core package.

The CPUID instruction provides also information about the maximum number of addressable logical processors in the system. The return value is same as the number of logical processors in the system if the BIOS has not disabled any logical processors. This information in CPUID leaf 1 can not be reliable in the physical machine due to configuration the BIOS. In the virtualized environment which BIOS configuration does not change, the return value of EBX[23:16] is equivalent to the number of logical processors in the system. The software can rely on this value in order to develop multicore computing on the system.

In a multicore processor, the number of cores has particular importance in multicore computing capacitance. Execution of a task on a logical processor, which its sibling is idle is faster that a logical processor that its sibling executing a task too. This is due to shared resources among two logical processors in the one core. Therefore, the number of cores helps software in order to determine how many logical processors share the resources on each core.

In the physical processor, each core is usually contains two logical processors, but in the virtual environment, it is different. In the virtual environment like QEMU, the number of logical processors and cores can be different as its physical processor. For instance, a QEMU virtual machine can receive 2 cores and 4 logical processors in each core through “-smp” directive. The logical processors will be same as the 4 cores with 2 logical processors in each core. Hence, the operating system can use the number of cores as a way to accelerates the computing performance.

The return value of EAX[31:26] of CPUID leaf 4 returns the maximum number of addressable core IDs. In the virtualized environment, it is equivalent to the number of processor cores in the system. The number of
cores along with number of logical processors used in order to enumerate processor topology.

In the multicore processors, the APIC is the heart of multicore computing. This interrupts controller unit, can wake up the other processor in the system and enable the inter-processor interrupt (IPI). All the MP and ACPI tables are decomposed in order to find information about APIC and its memory mapped registers.

Each logical processor in the system has APIC unit and accordingly APIC ID. The APIC ID and APIC memory mapped registers are key elements in order to develop multicore computing in the system. Each APIC unit has an ID so-called APIC ID so that other processors can send IPI to the destination APIC ID.

In the system that MP tables use the default values, the APIC IDs begin from zero to maximum logical processor in the system. Consequently, BSP gets the ID zero, so that the return APIC ID in the EBX[31:24] returns zero. Otherwise, the operating system can change the APIC ID of the APIC units, but it is not recommended. The APIC ID is an 8 bits ID constructed based on the processor topology. This means that system can address only 255 logical processor in the system so that system could not use more than 255 processors.

In order to address more than 255 logical processors in the system, x2APIC should be employed and accordingly system uses x2APIC ID. The main advantage of x2APIC is to use 32 bits ID to address the APIC units so that more than 255 logical processors can be addressed in the system.

The APIC memory mapped registers enable software to configure APIC and send an IPI to other local APICs. There are 25 APIC registers for each local APIC so that each logical processor has unique registers. The software can change the APIC memory mapped registers regardless of other APIC units. For instance, each APIC unit has its own timer so that it cannot conflict with the other logical processor in the system.

The key registers to wake up the application processors is ICR low and high registers which enable the local APIC to send an IPI to another APICs. Indeed, the ICR is a 64-bit register divided into two 32 bits low and high registers at the offset of 300h and 310h from base address respectively. The low-order 32-bit is the configuration register while the high-order 32-bit register contains the destination APIC ID or logical address.

The low ICR determines the vector number of interrupt, type of the interrupt, type of destination addressing, the type of signal and type of message delivery. The vector number determines the interrupt vector which the destination processor should send to its IDT for identifying the handler. In the STARTUP IPI, the vector number refers to the memory location in which the destination logical processors should begin to execute self-configuration code. Since the vector field in the low ICR is 8 bits so that it can only address the 4-kilobyte segment boundary on the STARTUP
Therefore, the *Intel MultiProcessor specification* emphasizes that the self-configuration code should start at the 4-kilobyte segment boundary, otherwise the logical processor can not execute the code.

The destination shorthand field in the low ICR, determines the type of message delivery. It can send the message to the particular APIC or broadcast on the system bus so that every local APIC connected to the bus receives the message. In order to wake up all the application processors at the same time, the all excluding self shorthand used during development. This happened while the sender is the bootstrap processor and all other application processors are not initialized. In this situation, the destination field in the high ICR is ignored. If all bits in the destination field set, then it is same as all including self shorthand.

While the no shorthand message delivery selected in the physical destination mode, the destination field in the high ICR determines the destination APIC ID. Although the destination field is 8-bit wide, but it can only address one APIC ID at a time. Otherwise, the software should use the logical destination mode addressing. In this mode, the 8-bit destination field refers to message destination address (MDA) and depends on the value of DFR and LDR registers of the local APIC. The MDA contains the bitmask of the destination and may target several APIC ID as its destination. Whenever the source sends an IPI with logical destination mode, the local APIC units checks the MDA with their DFR and LDR registers and if they are the destination then accept the IPI. Hence, the logical addressing mode requires the DFR and LDR configuration in the application processors.

The difference between the physical and logical destination mode, refer to their target domain while the no shorthand delivery selected as destination shorthand. This means that in the logical addressing mode the destination could be a group of logical processors. If the target is one APIC unit then it is recommended to use physical destination mode.

After the destination is determined, the type of interrupt should be chosen in the delivery mode. In order to develop multicore computing, the operating system should wake up all the logical processors first. The process of waking up logical processors starts with the initialization interrupt. The initialization interrupt triggers when the delivery mode set to 101b as INIT interrupt source. The figure 4.8 demonstrates a low ICR while it is configured to initialize all the application processors in the system. In the INIT IPI, the vector is ignored by the destination, so that it set to zero.

After sending INIT IPI to APs, each AP starts to initialize its own registers and set them to the default value. Therefore, it takes, the less than second in order to set all registers to their default. This means that after INIT IPI, processors reset to their default values and start in the real addressing mode. The INIT IPI should follow by the STARTUP IPI in which all the application processors wake up and ready to compute tasks.

According to *Intel MultiProcessor Specification* [66], the bootstrap processor
should make 10 mSec programming delay before sending STARTUP IPI to application processors. There are different methods of making programming delay such as creating a programming loop. But some compilers like Clang and GCC optimize loops and do not make enough delay. Therefore, the system timers are a reliable option to make programming delay. The PIT is a common solution to make delay or measure time in the system. In the development phase, the PIT channel 2 selected as a source for generating a delay between INIT and STARTUP IPIs.

The PIT channel 2, creates delay based on the delay between input and output of the channel 2 timer. In this method, whenever the input gate changed the output state changes with delay and this amount of delay is enough for application processor to initialize themselves with default values. But the programming delay can also be achieved by a couple of Nop Assembly instructions.

Although NOP Assembly instruction makes no changes to the registers and memory, but it may require a specific number of processor cycles. It is almost same as exchanging the EAX register with itself that has no effect on the registers but require processor cycles. During the experiments the PIT channel 2 had a conflict with the UDP timers so that the delay mechanism with PIT has changed with a couple of Nop instructions. Since the Noop instruction requires processor cycles, thus the amount of delay it creates has direct relation to the processor frequency. Therefore, several Nop instructions should be used in order to create required delay regardless of processor speed.

The STARTUP IPI wakes up the application processors and starts them in real addressing mode so that the application processors can access the low memory area through segmentation. In order to configure application processors to access whole the memory, they should configure to jump to protected mode through self-configuration code so that it named trampoline code. In the STARTUP IPI, the vector field contains the vectorized address of the self-configuration code. The code should be placed at the beginning of 4-kilobyte segment boundaries so that the 8-bit vector can address it for target application processors.

The self-configuration code is written in Assembly language and whole the program codes should be treated as text section so that processor executes only codes. Since it placed in the text section, thus it configured to start at the specific address which the STARTUP IPI referred. Therefor ORG instruction has been used in the self-configuration code to align it to a particular memory address. This shortlists the compilation output only to binary so that NASM can not compile the self-configuration code to ELF object file. Consequently, it makes the linking process complicated.

The self-configuration code must determine the GDT and IDT for each processor, otherwise application processors cannot jump to the protected mode. The GDT and IDT could be redefined in the self-configuration
code apart from the bootstrap processor but it is recommended to use the one GDT and IDT system-wide. The development used the existing system GDT and IDT in order to handle memory and interrupts likewise the bootstrap processor does.

In the real-addressing mode, each application processor increment counter in the memory to make sure that all the application processors are woken up. Since all the application processors may want to increment the counter in the memory at the same time, the counter become a critical region. The Intel MP specification [66] recommends that self-configuration code should employ a semaphore mechanism in order to deal with the race condition. But the semaphore creates new problems in the virtual environment and will be discussed later in this chapter.

After real addressing mode configuration, the processors should jump to the protected mode in order to access whole memory. This is a straightforward task by setting the bit zero of the CR0 control register. The processors may require an amount of time in order to set the CR0 protected mode flag. Hence, it is recommended to make some programming delay after that to make the jump secure. This programming delay can be implemented by Nop instruction.

In the protected mode, the application processors should run on the kernel mode so that they should fill the code segment and data segment with GDT 0x08 and 0x10 selectors respectively. The data segments require only "mov" instruction but code segment should be filled through a far jump. Thereafter, application processors are ready for executing the task.

Since the task execution process requires the stack and many languages like C++ use the stack heavily so that each application processor should have its own stack. By default Ubuntu 14.04 stack size is 8 MB in a machine with 8 logical processors. In this project the default stack size has determined 4 KB for each logical processor. The stack size has no standard size and each operating system, configure the size to meet its requirement. The Ubuntu 14.04 operating system has allocated bigger size memory to its stack so that its memory requirement for a virtual machine with more than 100 VCPU would be higher. In the development process, the 4 KB stack size decided because the system is under development and assumed that it will not execute the task with many variables which require more temporary space.

The stack definition is a straightforward process and it requires that software fills the ESP stack pointer with the address of a piece of memory. In the multicore system, each logical processor requires its own stack, with a unique stack pointer which has no conflict with others. Since the self-configuration code is following the single program multiple data (SPDM) technique so that it requires to separate the logical processor's stack implementation. This thesis has used the APIC ID to allocate a unique address to the stack pointer of each logical processor.

After stack allocation to each application processor, they ready to execute
tasks. Hence, each application processor calls the nth prime number calculation function and executes it as a parallel computing task. Each application processor executes its task and requires to store the result.

The returned results should be stored in the memory so that the operating system can read them and service the request. There are two methods for sending requests to the application processors and receiving the results. One method is message passing that the application processors inform the BSP that the task is finished and the results stored in the memory. Another method is shared memory method in which the request and the results stored in the specific memory locations.

The BSP can send a request to the APs through memory queues and retrieve the results through memory arrays. The BSP and APs share the memory with each other and they use APIC ID to identify the producer and consumer. Accordingly, the proper producer and consumer algorithms are necessary in order to prevent deadlock.

The development has achieved the multicore computing based on the existing IncludeOS operating system. This development gets the processor information and used the APIC in order to send an IPI to APs. The APs configured themselves through self-configuration code in order to execute given task and share the results through the memory with other processors in the system.

5.2 Critical Section in Multicore Operating System

In the real life whenever the number of participants increases, the possibility of the conflict increases too. This scenario may happen in the operating system such that The resources that are shared in the system are the possible sources of conflict and called critical sections.

The operating system must avoid the concurrent access to critical sections so that prevent the possible conflict. Indeed, the concurrent access requires mutual exclusion. There are different types of mutual exclusion such as semaphore, monitor, locks and message passing. The semaphore is widely used such that the Intel® 64 and IA-32 Architectures Software Developer’s Manual suggested to use semaphore while incrementing the counter in the self-configuration code. Hence, it introduces the race condition such that the different parties compete over a shared resource.

In the multicore operating system, the race condition is inevitable due to multiple processing units that operate concurrently. Each core may compete over the same shared resource and it introduces the race condition. Hence, the operating system should deal with race condition over a critical section.

In the development, application processors should increment the counter while running the self-configuration code. Since the application processors
have woken up through a broadcasted IPI thus they run the self-
configuration code concurrently. It introduces a new problem so that
operating system should deal with it otherwise the data in the shared
memory may be incremented unpredictable. The regular option to handle
this situation is to use semaphore lock in order to allow one application
processor access the critical section at a time. But this also introduces a
new challenge in the virtual machines.

One application processor holds the lock and want to write to the critical
section but at the same time hypervisor pause it and allocate the physical
processor to another application processor. In this situation, the other
application processors should wait until the one which holds the lock
return from the pause and release the lock so that other application
processors could use the lock. Indeed, this will waste the processor cycles
in the multicore operating system.

This research used another method such that it used the LOCK instruction
while incrementing the counter. The LOCK instruction causes locking the
bus so that the underlying hardware manage the race condition in order
to lock the bus and make the instruction atomic. The processors connected
to the bus use usually low priority mechanism in order to deal with race
condition over the bus acquisition. Hence, the race condition over the
critical section will be managed by the hardware so that it avoids the new
challenges in the virtual environment. Also, the locking the bus simplifies
the development of multicore computing in the operating system.

5.3 Multicore Communication

The processor may require communicating with other processors in the
system. Processors in a multicore system should be able to send a request
to other processor and receive a reply from. The communication among the
processors can occur through shared memory or message passing.

The shared memory is a common method in a multicore system such that
the implementation is straightforward. In the shared memory technique,
one processor writes to a specific memory location and the other read it.
With this technique, the producer and consumer should agree on a specific
memory location for data exchange but it introduces a critical section.
On the other hand, the message passing technique relies on messages
and message transferring techniques. The message passing technique is a
challenging method to distribute message among the nodes. Indeed, in the
distributed computing the message passing is inevitable, but in a multicore
system, it will be much simpler to use the shared memory technique.

During the multicore development process, the shared memory has been
used in order to reduce the complexity of the development. Although
the number of logical processors may increase to more than a 100 in a
virtual machine, but they are in the same system and can use the shared
memory. During the development, the application processors used the shared memory to store the results of their execution and made an array in the memory.

Since the operating system follows non-preemptive task execution, thus each processor require storing a limited amount of information in the memory. In this research, each application processor stores its execution result to a particular memory location which identified by the APIC ID. The consumer which is the BSP check the particular location for a new data. Indeed, the BSP does a busy waiting in order to check whether the producer has written a data in the agreed memory location and the producer which is AP write to the location. With this mechanism, the producer shares its execution results with other processors.

Here, it is assumed that the BSP plays always a managing role and is the only consumer in the system. This assumption simplified the producer and consumer problem in this development. The development considered the maximum number of 255 memory location in order to store the results in the memory. This is due to that development can address maxim 255 APIC ID in the system consequently 255 logical processor in the system may exist.

The busy waiting for a memory location in the multicore system is not an efficient method. The processor may have a feature to monitor the memory location, consequently react to any change in the state of the monitored location. The MONITOR feature is a hardware monitoring mechanism in which checks for store operation in the memory. Indeed, the producer should store the data into memory using the STOS instruction in order to MONITOR instruction can detect the memory write. The store to a memory address triggers the monitoring hardware. On the other hand, the MWAIT feature provides hints to the processor to enter an optimized state. The MONITOR/MWAIT instruction eliminates the busy waiting and cause the processor to enter an optimized state while waiting for a change in a memory location.

The software can check the processor features for hardware monitor through CPUID instruction. During the development process, the return value of ECX[3] in CPUID leaf 1 demonstrated that the host processor supported the hardware monitor, but the QEMU does not enable this feature for the guest processor even though the KVM is enabled. Indeed, the virtual environment did not support the hardware monitor so that the busy waiting solution selected in the development process in order to read the APs’ execution results from shared memory.

5.4 Non-Preemptive Multicore

The preemptive task scheduling has an impact on the performance of the system. The regular preemptive task scheduling enables the operating system to manage and execute multiple tasks with limited resources.
Indeed, the preemptive operative system shares the limited resources between the multiple tasks such that each task receives a fair amount of resources. On the other hand, the non-preemptive operating system allows the task to use the resource until it finishes. In this method, other tasks which waiting for the processing resource may wait for a long time which is called starvation.

During the development, the non-preemptive task execution was selected as a task execution mechanism in a multicore environment. The virtual environment can create as many resources as the operating system supports. This applies to processors as well, such that the hypervisor can subscribe as many processors as the guest operating system can manage. If the host system has 8 logical processors then it can subscribe as many virtual processors as guest operating system can support. Indeed, assigning more processor than physical, logical processors called over subscription of the physical processor.

The over subscription techniques make the hypervisor do preemptive scheduling among the physical processors. If a virtual machine has assigned many processors, then the hypervisor must do preemptive scheduling. This is called context switching in the hypervisor level. Since the hypervisor can context switch, thus the operating system can follow non-preemptive task execution in order to reduce the overhead.

This research has followed the non-preemptive task execution while over subscribe as many processors as possible to the virtual machine. This introduces a new scheme to configure virtual machines to use as many virtual processors as they require in order to execute tasks in the system.

5.5 Development Evaluation

The evaluation of the development is a final step in each project. It helps the users or other researchers to judge the development. Indeed, the evaluation phase assesses the development based on a set of particular methods. There are different methods for evaluating the development. An evaluation method may assess the input and output data, observe the development in the actual use or evaluate of efficiency.

The two first methods are not applicable to this developmental research because the IncludeOS is under development and is not fully functional. But the efficiency of development can be evaluated in comparison to other solutions. The evaluation of a development requires the assessment of particular development.

Since the research motivation was to enhance the efficiency of the existing system through multicore computing thus the efficiency can be a key factor for evaluation. The efficiency is generally measured as a ratio of output to total input. But in computer science, it is difficult to calculate the output
and input with numerical values such that the ratio consequence to the efficiency.

The efficiency in computer science can be calculated through surrogate variables such as time and cost. The surrogate variable provides the indirect measurement of parameters in which direct measurement is not possible. In this research, the surrogate variables are time and cost which help the measurement of efficiency. Here, the execution time of the task can be measured through experiment. On the other hand, the cost of ownership can be calculated using the cost of processors, memory and disk that each solution uses.

In the evaluation phase, the multicore IncludeOS compared with the multiple single core IncludeOS, bare metal Ubuntu 14.04 and Ubuntu 14.04 installed in a virtual machine. In the experiment, 36 IncludeOS instances booted up in comparison to an IncludeOS with 36 virtual processors. On the Ubuntu operating system, the experiments performed through Pthread model in order to execute 36 tasks concurrently. The Each experiment used the same nth prime computing as a processor intensive task and send the task to the client through a network protocol.

In the previous chapter, the cost of ownership described as the cost of memory and disk which each solution requires in order to perform a specific task. Because the number of processors is same in both solutions and the memory and disk usage differs among the solutions, therefore the 36 IncludeOS instances cost 36 times more memory and disk in comparison to multicore IncludeOS. Hence, the cost of ownership of the multiple single core IncludeOS would be definitely higher than one multicore IncludeOS whenever a number of tasks are more than one. On the Ubuntu operating system, the cost of memory and disk is more than Unikernel operating system because the Ubuntu require more memory and disk in order to be functional.

On the other hand, the execution time is another surrogate variable in order to measure the efficiency indirectly. Often the time factor refers to a parameter for performance measurement. Here, the time is a total time of the sending task, task execution time and reply the answer to the client. It is obvious that the round trip time of transferring request and reply is overhead within the measured time. In the multiple single core IncludeOS, the total time is the round trip time of sending tasks, task execution time, reply answer and summation of results. However, all the experiments followed the same procedure so that they produced valid output.

The experiment procedure was restricted to the existing system functionality. Since the IncludeOS project has no console for user interaction and the single method of communication is through the network thus experiments restricted to network communication. Among the two main network protocol, UDP selected in the experiment because IncludeOS has a simple UDP service so that the UDP connection was stable. On the other hand, the TCP protocol was under development so that it did not establish a stable con-
The Ubuntu operating system follows the same procedure during the experiment.

The experiments used a bash script in order to run 50 identical experiments for each solution. The experiments repeated 50 times in order to demonstrate that the collect data is predictable. Hence, a couple of scripts ran the experiments and measured the computation of the 500000th prime number. The table 5.1 demonstrates computing time of 500000th prime number among the different solutions.

The table 5.1 demonstrates that the mean time of multiple single core IncludeOS instances is 5.22 seconds. This amount of time consists of the execution of 500000th prime number and sum the results such that the script measured the time which client received the 36 results and summed them. The scripts ran the 36 instances of IncludeOS and sent the request to the IncludeOS instances and accordingly it summed the results and returned the one value.

On the other hand, the multicore IncludeOS with 36 virtual processors received the request through the network and woken up the 35 application processors to execute the tasks along with bootstrap processor. The multicore IncludeOS which equipped with 36 virtual processors, took approximately 4.28 seconds in order to execute 36 given tasks and sum the results so that it returns one value. The bare metal Ubuntu executed 36 tasks in 4.97 seconds and the Ubuntu installed in the virtual machine executed 36 tasks in 6.21 seconds. The table 5.1 demonstrate that the total measure time of the multicore IncludeOS is less than other solutions.

<table>
<thead>
<tr>
<th></th>
<th>Measured time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multicore IncludeOS</td>
</tr>
<tr>
<td>Mean</td>
<td>4.28</td>
</tr>
<tr>
<td>95% confidence interval</td>
<td>4.25-4.3</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time of 500000th prime number computation within different operating system with 36 cores

The table 5.1 demonstrates also the 95% confidence interval of the measured time. The confidence interval explains the amount of uncertainty related with a sample estimate of a population parameter. Hence, the table 5.1 demonstrates that the measured times have small variation and the outputs are predictable due to the stable experiment condition.

The multicore IncludeOS allows the application processors to be in halt mode while there is no request for them so that they consume no power so that other virtual machines on the hypervisor can use the maximum processor power. Indeed, this helps physical processors’ efficiency of the host machine. Therefore, the number of assigned processors to the multicore IncludeOS has no effect on the other machine until a new request performs the multicore computing.

On the other hand the single core IncludeOS initializing the bootstrap

105
processor during the operating system boot up process. In overall both solutions use the same amount of processor time.

In regard to memory and disk space, each single core IncludeOS requires dedicated memory and disk while the multicore IncludeOS can provide better resource utilization. Indeed, the multicore IncludeOS will be more efficient than multiple single core IncludeOS instances and Ubuntu operating system. On the other hand, the cost of ownership of the multicore is less than a multiple single core IncludeOS instances so that the cloud service will be cheaper with multicore computing.

5.6 Summary

This chapter analyzed the multicore computing development process and argued the decision has been taken during the development. Afterward, the problem statement question took under consideration and argued such that how the development addressed them. In the final step, the development evaluated in comparison with the multiple single core operating system, bare metal Ubuntu and Ubuntu installed in virtual machine through analysis of the undertaken experiments.
Chapter 6

Discussion

This chapter discusses the different aspects of the development and suggests improvement for future work. The chapter begins with an overview of the whole study and achievements and follows by discussing the data generation and results. Later the future work will propose further research projects.

6.1 Research Master Plan

This research has developed a service in order to provide multicore computing for an existing Unikernel operating system. The service has developed multicore computing for IncludeOS operating system in which it was the single core and single threaded operating system. Therefore, the IncludeOS now take advantage of the multicore computing in a virtual machine.

The research demonstrated that the local APIC is a key unit in order to develop a multicore operating system. Since each processor in the system has dedicated APIC so that it is called local APIC. Hence, the advanced programmable interrupt controller (APIC) registers are tools for configuring the APIC units. The local APIC of the bootstrap processor sends the INIT-STARTUP IPI sequence to the target application processors in order to wake them up.

Each processor has a set of memory mapped APIC registers in which they are identical in functionality among all the processors connected to the system bus. The local APIC registers are 32, 64 and 256-bit registers that larger registers can be treated as several 32-bit registers. The registers, handle all the interrupts directed to the processor and can redirect or discard the interrupts. But there are two inter-processor interrupts in which the local APIC reacts regardless of configuration. The INIT and STARTUP IPI sequence will send a soft reset signal to the target processors.

The bootstrap processor should use its own local APIC in order to send
INIT-STARTUP IPI sequence to other processors in the system. This requires that operating system determines the base address of the APIC registers in the memory. Hence, software should retrieve information about system processor in order access the controller units in the system. The main source of information is the BIOS which is a type of firmware used to initialize the hardware devices.

In the virtual environment, the virtual hardware initializes also with the BIOS which is an emulated one. In this research, the virtualization platform consists of KVM/QEMU combination so that the BIOS in the virtual machine is also emulated. The QEMU emulator uses the PC BIOS from Seabios project and the Plex86/Bochs so that the virtual hardware initialize by them. The development demonstrated that the OEM ID of the BIOS specific table in the memory is Bochs.

The BIOS build a series of tables on the memory for the operating system so that the operating system can retrieve hardware information from them. The main informative table in the multicore system is that MP configuration table which includes a chain of tables. Since the BIOS has not standardized yet so that each BIOS use a particular part of memory in order to build informative tables. Even though that Intel MultiProcessor Specification [66] presents that the MP configuration table should place at the EBDA or end of the base memory or the first kilobyte of the BIOS area, but the development searched the BIOS area in the low memory in order to find the MP floating pointer signature accordingly the MP configuration table.

According to Intel MultiProcessor Specification [66], the existence of the MP configuration table demonstrates that the system is multiprocessor compliant. Hence, one method to check if the system is multicore is to determine whether it contains the MP configuration table. On the other hand, the MP configuration tables should be updated by the operating system otherwise it contains just the default values written by the BIOS. The development demonstrated that the MP configuration tables in the QEMU emulated virtual machine contains only the default information. Since the operating system requires developing the multicore computing so that the MP configuration table information can be achieved through other procedures. But development can update the MP configuration table for later use by the operating system.

On the Other hand, ACPI tables are another source of information in which BIOS creates. These chains of tables consist of not only multiprocessor information, but other information as well. The ACPI specification [41] presents 25 tables in which they contain hardware information. The development found that the emulated virtual machine by QEMU contains only 6 tables instead of 25 tables. Since the QEMU uses the Bochs as its BIOS emulator thus the OEM ID of the tables is Bochs. The ACPI consist of 4 extended tables such that the MADT presents the APIC information. In the ACPI chain of tables like MP configuration table contain only default information.
The development found out that the APIC related information in the ACPI and MP configuration tables can be achieved through IA32_APIC_BASE model specific register (MSR). This MSR designed as architectural MSR in which its functions remain same for succeeding families of IA-32 processors. This MSR is unique for each logical processor such that it provides information about each processors’ local APIC. Since the MSRs have the fixed address and software can read and write to them by RDMSR and WRMSR Assembly instructions respectively, thus it used during this development in order to get information about the base address of local APIC register instead of the ACPI and MP configuration tables.

On the other hand, the CPUID instruction returns the processor information in the general purpose registers. This instruction returns information about the processor that execute the CPUID so that executing this instruction in the first step returns the bootstrap processor information. During development the CPUID instruction used in order to detect Hyper-Threading Technology. If the processor supports the Hyper-Threading Technology feature, then it is a multicore processor. In regard to this, the development considered that if the processor supports the Hyper-Threading Technology then the system is multicore compliant. Hence, the CPUID instruction can be used in order to determine whether the system is multicore compliant instead of using MP configuration table.

The CPUID instruction returns more information about the multicore system. It returns information about the number of maximum addressable cores and logical processors in the system. In the virtual machine, the development found out the maximum addressable cores and the logical processors returned by CPUID is same as the number of cores and logical processors in the system. Hence, the software can rely on the return value by the CPUID in order to count the number of available cores and logical processors in the system so that it can enumerate the processor topology.

After retrieving the information about the multicore system and the APIC then the software can configure the local APIC register in order to send a wake-up signal to the application processors. The local APIC of the bootstrap processor sends the soft reset signal to the application processors first. Thereafter, the bootstrap processor requires performing programming delay in order to allow application processors to reset their registers to a default value. The development used a PIT unit in order to perform delay, but development tested also the Nop Assembly instruction in order to perform delay.

After sending the INIT IPI to application processors and allow them to reset their registers to a default value, then local APIC sends a STARTUP IPI with a pointer to the memory location which the self-configuration code resides. Indeed, STARTUP IPI forces the application processors to execute self-configuration code stored in the 4-kilobyte boundary in the low memory in which vector field points to there. The self-configuration code which is called trampoline is same as boot loader code for the bootstrap processor.
Since the application processors start in the real addressing mode, thus the self-configuration code should determine the global descriptor table (GDT) and interrupt descriptor table (IDT) for each application processor and they increment the counter to show that they are alive. Afterward, application processors jump to protected mode in order to run in a kernel mode and access whole the memory.

In the self-configuration code, the stack pointer of the application processors should be initialized with the address of their dedicated stack. Hence, the application processor could store temporary data into stack during the task execution.

Thereafter, the application processors are ready to execute tasks and can jump to a specific address and accordingly execute the code. Finally, the application processors run their tasks and store results in the shared memory so that the bootstrap processor can use the results. In order to be power efficient, the application processors enables the interrupts and change their state to halted.

6.2 Identifying Processors

The development used CPUID instruction in order to enumerate the cores and logical processors. The testing phase demonstrated that the maximum addressable cores and logical processors returned by the CPUID are same as the number of available cores and logical processors in the virtual machine. The APIC ID is an identification number for each local APIC in the system.

The APIC ID is a numeric ID which is started from zero to maximum 254 in case that the system uses the xAPIC architecture. The development used the APIC ID in order to identify different processors in the system. Accordingly, development used the APIC ID to build an array for the processors in the memory so that each processor uses its dedicated stack and shared data in the memory. Indeed, it used APIC ID to do indexed partitioning in the memory in order to utilize the shared memory technique in the multicore system.

On the other hand, the APIC ID is used to address specific processor in the inter-processor communication. Hence, the target processor identified by its APIC ID and the local APIC uses the destination APIC ID in its ICR registers in order to specify the destination of IPI.

The CPUID instruction returns the APIC ID of the processor that executes the CPUID instruction. In the self-configuration code, each application processor executes CPUID instruction such that they identify their APIC ID and accordingly use APIC ID to find their allocated memory portion.

The processor identification helps the development to partition memory for each processor in the system and manage them through their ID.


6.3 Race Condition Among Processors

The multiple processors may race with each other over a shared resource. The development used atomic operation in order to steer the race condition management to the hardware. Indeed, self-configuration code utilized hardware in order to handle race conditions in the multicore system.

The race condition can be handled by semaphores during the development. But semaphore in the virtual environment causes new challenge. The hypervisor may pause the processor, which holds the semaphore so that the other processors can not use the semaphore until the paused processor become operational again and release the semaphore. Therefore, development used the LOCK instruction in order to hardware deal with the race condition.

The LOCK instruction ensures that the processor has exclusive access to the shared memory. This is done by the hardware so that the above mentioned problem could not happen. However, software should use the common lock based mechanism in order to deal the race condition.

In comparison with related works, the OSV project avoids also the Spinlock in OSV Unikernel operating system. The OSV operating system is Spinlock free operating system, but they did not mention how they avoid using the Spinlock in their operating system. This research managed the race condition through using the atomic operations and locking the bus in the shared memory situation.

6.4 Task Management

The operating system requires a task management mechanism such that it defines how to perform the multiple tasks and accordingly store the results. This research used the non-preemptive task management mechanism in which the operating system performs the task until it finishes then stores its results in the shared memory. In this mechanism, the other tasks will not interrupt the running tasks until it finishes.

On the other hand, the preemptive task management mechanism shares the processing units with multiple tasks so that the running task may be interrupted some time. In this mechanism, the operating system can interrupt the running task and share the processing unit with other tasks so that each task has a fair amount of processing unit time.

This research used the non-preemptive task management mechanism while the operating system can employ multicore computing in order to perform multiple tasks concurrently. Indeed, the operating system may employ one core per task in order to perform the tasks and store the execution results into the shared memory. On the other hand, the non-preemptive
development is a simple solution as task management mechanism among the multiple cores.

6.5 Shared Data

In the multicore system, the processor may require the results of the other processors in the system. The development, used shared memory technique such that each processor may read the results of the other processors. The development used shared memory in order to implement single program multiple data (SPMD) technique.

The SPMD technique will decrease the duplication and generate data based on a single program. The development used this technique in the self-configuration code such that processors run the same self-configuration code and generate their own data. Hence, the multicore system requires a shared memory technique in order to access the single program at the same time.

The shared memory technique reduces the access time, but on the other hand is not scalable with the fixed memory size. Indeed, each processor requires a portion of memory, such that the many processors require many memory portions. Hence, the memory should be increased in regard to the number of processors in the system. Otherwise, the operating system gets out of memory in which the performance decreases.

In contrast to the shared memory technique, the message passing is a scalable technique such that perform send and receive actions. In the massage passing technique, each processor has its own address space. Although the message passing technique is scalable, but it requires message communication and connection-oriented communication between sender and receiver. On the other hand, the message passing technique requires dedicated memory space likewise the shared memory. Hence, the system allocated more processors, thus it requires more memory regardless of the processor communication techniques.

The shared memory technique is common in the multicore processor in which use the uniform memory access (UMA) so that each processor access the memory at the same time. In the nonuniform memory access (NUMA) each processor accesses its own local memory faster than the local memory of other processors. Therefore, the performance of the shared memory technique in the NUMA nodes reduced.

The development used shared memory technique in such way that it facilitates the development process and reduce the complexity of multicore computing.
6.6 Multicore Versus Multiple Single Core Operating System

This research developed multicore computing based on the IncludeOS Unikernel operating system. The development provides the IncludeOS with maximum 255 processors so that it can compute in parallel. This enhances the efficiency of the IncludeOS in which IncludeOS take advantage of the multicore computing.

On the other hand, multiple single core IncludeOS will increase the parallelism. Although the multiple instances of IncludeOS increase the parallelism, but it requires more resources so that the cost of the multiple instances will be more than multicore IncludeOS.

The experiments in the previous chapter demonstrated that the cost of multiple instances of operating system for \( n \) tasks will be \( n \) times the memory and disk more than the multicore operating systems. Hence, the cost of ownership will be more than the multicore operating system.

On the other hand, the multicore IncludeOS requires extra time to wake up the cores in which they can be ready for task execution. Although it requires waking up time, but the multiple cores have access to shared data inside the operating system so that it compensates the wake-up time. Whenever the tasks require the further computation then it performs inside the operating system because the cores share their data. The results demonstrated that multicore IncludeOS performs better than other solutions, especially with multiple single core IncludeOS.

Since the multiple single core IncludeOS can not share data with each other thus the data should be shared on the other palace and extra computation performs on the third party machine. This makes the execution time longer than multicore computing while the cores inside the operating system share data among themselves. Therefore, the performance of the multicore operating system increases whenever the cores require extra computing with the shared data.

On the other hand, the experiments demonstrated that the multicore IncludeOS execution time is less than an Ubuntu operating system so that the multicore Unikernel operating system performs faster than the Ubuntu 14.04 operating system.

On the other hand, the MirageOS project wants to provide an efficient runtime for single core computing with a common immutable data store so that a large cluster of cloud-based virtual machines operate over data. In this solution, the virtual machines will share the data instead of the processors. Mirage project aims to run the clusters of MirageOS through multiscale compiler support in order to adopt the communication model with hardware platform’s constraints [85].

As mentioned above, building the multiple operating in a cluster requires
more resources and accordingly increase the cost of ownership. On the other hand, sharing the data between the virtual machine will introduce the new challenges in the aspect of implementation and the security. Otherwise, the multicore operating system will generate the same level of parallelism with fewer resources and it facilitates the shared data techniques.

To conclude, this research demonstrates that the multicore computing enhances the efficiency, functionality and performance of Unikernel operating system in comparison to multiple single core Unikernel operating system and Ubuntu operating system. The results confirmed that the Unikernel operating systems will address the shortcomings of the regular operating systems on the cloud. Hence, the multicore computing enhances the Unikernel operating system functionality, performance, and efficiency in order to compete with other existing solutions.

6.7 Future Work

This research opened several channels for further investigation and research. Since the Unikernel operating systems are under development so it is open for new ideas and research which due to time restriction, they must be done as a future work. This research developed the multicore computing for the IncludeOS Unikernel operating system and due to the broad field of research in the multicore computing thus more investigation can be done in order to evolve it into a robust Unikernel feature.

In order to evolve the multicore computing, the future work can develop a scheduling system for task distribution between application processors. Hence, the bootstrap processor can receive tasks through the network and send it to the application processors for execution. This kind of scheduling leads to efficient use of application processors so that the BSP push the application processors to change its halted mode and execute the task.

The I/O APIC is responsible for external interrupts and directs them to the local APIC. Since the Unikernel operating system communicates through the network, thus the huge amount of external interrupts generated by the network interface. The future research can investigate the I/O APIC in order to increase processor affinity by distributing the external interrupts to different processor So that the external interrupt does not overwhelm one processor.

The xAPIC feature provides a 8 bits APIC ID in which enables the operating system to address maximum 255 processors. The future research can develop the multicore computing with x2APIC such that the operating system can address more than 255 processors in the system. This leads the operating system to manage manycore processors in the system in order to provide much more computing power to Unikernel operating systems.

The hypervisor provides hot-plug processor to the virtual machine such
that the virtual machine can detect the added processor without rebooting. Some of the hypervisors such as Xen provides the hot swap processor to the virtual machine such that the processor can be added or removed from the virtual machine without rebooting. The future research can develop a solution in which add or remove the virtual processors to the Unikernel operating system in order to adopt the operating system with incoming workload. Hence, the non-preemptive operating system can take advantage of oversubscribed processors in the hypervisor level and remove the extra processors whenever the workload is low so that other virtual machine may require processors.

6.8 Summary

This chapter discussed the development process and the decision made during development. Then, it discussed the research question and how the questions have been answered during the development process. Finally, it proposed the ideas so that they can be investigated in the future work.
Chapter 7

Conclusion

The multicore processors are now standard architecture in the market such that every system uses the multicore processor as central processing unit. The processor vendors produce much faster processors with multiple and many cores inside so that it enhances the processing power of the whole system. Thus, the cloud-based Unikernel operating system should also evolve in order to take advantage of multicore processors.

This research developed a multicore computing for IncludeOS Unikernel operating system. The research built a new service based upon existing IncludeOS Unikernel operating system which it is also under development. The development answered the research questions raised in the first chapter and some of the encountered problems during the research suggested to be answered in the future work. The development of multicore computing started with building the existing system which is the IncludeOS and do exploration about its architecture. Using the IncludeOS as existing system make the development follow its system design meanwhile developing an independent service which can be treated as a module.

The project as an experimental research followed and iterative process of exploration, solutions, development, test and evaluation. The research investigated the existing system and the processors architecture. The exploration phase caused the solution proposal which led the research to achieve its aim. In the development phase, each solution implemented with program codes and followed by the testing phase which tested the results to determine whether the output is as expected. This portion of the research was in a permanent iteration until it satisfied the research aim. Finally, the development evaluated with competing solutions in order to assess the whole project.

The research used the processor specifications such that processor features investigated in order to develop multicore computing. The investigation demonstrated that the local APIC is a dedicated unit in which it controls all incoming interrupts toward the processor. Thus, it used in order to
manage processor cores in the system. The bootstrap processor used the local APIC in order to send inter-processor interrupt (IPI) to application processors in the system. The bootstrap processor should send INIT IPI as a soft reset signal to application processors in order to application processors reset to the default value. After the soft reset, the bootstrap processor sends STARTUP IPI to application processor to direct them to run the self-configuration code in the memory and configure them to be operational for task execution.

The development process answered the research question such that it has woken the application processors up to develop a multicore computing. The development used the APIC ID in order to identify the processors such that each processor identified by its unique ID in the system. During the configuration of application processors, the other research questions answered. In the research, the race condition addressed through using the LOCK instruction in order to handle the race condition at the hardware level. The development managed multiple tasks in a non-preemptive manner and used shared memory technique such that application processors shared their data with each other. The research developed all these solutions and accordingly achieved the multicore computing within the IncludeOS Unikernel operating system.

The experiment's results demonstrated that the multicore service in the IncludeOS could execute multiple tasks simultaneously on a multicore processor. Hence, the developed multicore computing compared with multiple single core operating systems, bare metal Ubuntu and the Ubuntu installed in a virtual machine such that the same amount of tasks executed with competing solutions. The results demonstrated that the multicore Unikernel performed the multiple tasks faster than multiple single core Unikernel operating system, bare metal multicore Ubuntu and multicore Ubuntu installed in a virtual machine.

The experiments demonstrated that a multicore Unikernel operating system achieves better performance than a regular operating system such as Ubuntu. The multicore Unikernel operating system is a cost-efficient solution in which requires less memory and disk in comparison to other solutions. Therefore, multicore computing for Unikernel operating system enhances the performance while it saves the memory and disk.

Finally, the multicore computing enhances the efficiency, functionality and performance of the Unikernel operating system through increasing the processing units and reducing the memory and disk consumption. By utilizing this research, the multicore computing improves the efficiency, performance, and functionality of the Unikernel operating systems. Accordingly Unikernel operating systems can compete with existing solutions in the cloud environment.
Bibliography


Lawton G. ‘Moving the OS to the Web’. In: *Computer*, 41.3 (Mar. 2008), pp. 16–19. ISSN: 0018-9162. DOI: 10.1109/MC.2008.94


Appendix A

Program codes

Listing A.1: Multicore computing service

```c
#include <os>
#include <iostream>
#include <stdio.h>
#include <string>
#include <net/inet4>
#include <net/dhcp/dh4client.hpp>
#include <math.h> // rand()
#include <sstream>
#include <list>
#include <vector>

#define HW_HTT_BIT (1<<28)
#define APIC_BASE_MSR_addr 0x1B
#define APIC_ID_Version 0x30
#define ICR_low 0x300
#define ICR_high 0x310
#define GDTR_addr 0x80400
#define IDTR_addr 0x80440
#define APcounter_addr 0x80480
#define Trampoline_addr 0x80000
#define APIC_Error_Status 0x280

extern char _binary_trampoline_bin_start;
extern char _binary_trampoline_bin_end;
extern "C" {
    extern int prime(int id);
}

void AP_INIT(void);
uint32_t CPUIDsupported();
void _CPUID(uint32_t,uint32_t,uint32_t*);
std::string VendorID();
uint32_t HTT_supported();
int MaxLogicalProcessor();
unsigned int MaxcoreID_Package();
uint32_t InitialAPICID();
```
using namespace std;
void Service::start()
{
    int MAXInput=0;
    MAXInput=CPUIDsupported();
    *((uint32_t*)0xFEE00370)=0; // clear LVT Error
    cout<< "The processor Vendor ID: " <<VendorID()<<endl;
    cout<< "The maximum Logical processor ID per package: " " <<MaxLogicalProcessor()<<endl;
    cout<< "The Maximum cores ID per package: " " <<MaxcoreID_Package()<<endl;
    Processor_topology();
    AP_INIT();
}

//MP_table(); //Print MP table fields
//ACPI_table(); //Print ACPI table fields

} /////////////// Wake up the APs /////////////
void AP_INIT()
{
    uint32_t APIC_BASE_addr=0;
    uint64_t gdt=0, idt=0;
    APIC_BASE_addr=(_RDMSR(APIC_BASE_MSR_addr) & 0xFFFFF000);
    char* start =&_binary_trampoline_bin_start;
    char* end =&_binary_trampoline_bin_end;
    memcpy((void*)0x80000, start, end-start); // copy trampoline to 0x80000
    typedef int(*prime_t)(int);
    prime_t* prime_addr = (prime_t*)0x804c0; // Save the address of function at 0x804c0
    *prime_addr = &prime;
    asm("sgdt %0 ":="m" (gdt));
    asm("sidt %0 ");
// Store GDTR and IDTR in 0x804000 location
*((uint64_t*)GDTR_addr)=gdt; // save GDT address
*((uint64_t*)IDTR_addr)=idt; // save IDT address

uint16_t * APLIVE =(uint16_t*) APcounter_addr; // initialize AP counter
*(APLIVE)=1;

// Fire INIT
*((uint32_t*)(APIC_BASE_addr | ICR_high))=(1<<25); //← Write ICR upper dword and wakeup Vcpu2
*((uint32_t*)(APIC_BASE_addr | ICR_low))=0x000c4500; //← Trigger INIT interrupt

asm ( "in $0x61 , %%al \n" 
"and $0xFE , %%al \n" 
"out %%al , $0x61 \n" 
"or $1, %%al \n" 
"out %%al , $0x61 \n" 
".delay%: \n"
"in $0x61,%%al \n" 
"and $0x20,%%al \n" 
"jz .delay%=" 
:: 
); // make delay
/*
asm( "nop \n" "nop \n" "nop \n" "nop \n" "nop \n" "nop \n"
); 
*/

*((uint32_t*)(APIC_BASE_addr | ICR_low))=0x000c4680; //← Send SIPI with start address 0x80000
while(*APLIVE < MaxLogicalProcessor()){
} // Wait until all APs execute trampoline and ← increment counter

// Number of active processors in the machine
printf("Number of activated AP(Revenants): %d \n",((*← APLIVE)-1));

}
uint32_t EDX = 0;

if (CPUIDsupported() && VendorID() == "GenuineIntel") {
    asm ("cpuid" : "=d" (EDX) : "a" (1));
    if (EDX & HW_HTT_BIT)
        return 1;
    else
        return 0;
} else
    return 0;

// ////////// Determine whether CPUID is supported //////////
uint32_t CPUIDsupported(void) {
    uint32_t flagrd = 0, flagwr = 0;
    asm volatile("pushfd \n"
           "movl %%edx, %0 \n"
           "mov %%edx, %%eax \n"
           "xor $0x200000, %%eax \n"
           "push %%eax \n"
           "pop %%eax \n"
           "push %eax \n"
           "movl %eax, %1 \n"
           "push %edx \n"
           "pop %eax \n"
           "movl %eax, %1 \n"
           "push %edx \n"
           ":=m" (flagrd), ":=m" (flagwr) \n");
    if (!(flagrd ^ flagwr))
        return 0; // CPUID is NOT supported
    else
        return 1; // CPUID is supported
}

// ////////// Return the CPUID instruction //////////
void _CPUID(uint32_t eax, uint32_t ecx, uint32_t *reg) {
    if (CPUIDsupported()){
        asm ("cpuid" : ":=a" (reg[0]), ":=b" (reg[1])
             , ":=c" (reg[2]),
             ":=d" (reg[3]) : ":a" (eax), ":c" (ecx));
    } else{
        reg[0]=0; reg[1]=0; reg[2]=0; reg[3]=0;
    }
}
Detect Vendor Identification String

```cpp
std::string VendorID(void)
{
    uint32_t VID[3]={0,0,0};

    asm ("cpuid": "=b" (VID[0]), "=c" (VID[2]), "=d" (VID[1])
        : "a" (0)
        );

    std::string IDD = (char*) & VID[0];
    IDD = IDD + ((char*) & VID[1]) + ((char*) & VID[2]) ;
    IDD.erase (12);
    return (IDD);
}
```

Enumerate maximum addressable logical processor

```cpp
int MaxLogicalProcessor(void)
{
    uint32_t EBX=0;

    if (HTT_supported())
    {
        asm ("cpuid": "=b" (EBX)  
            : "a" (1)
            );
        return ((EBX & 0x00FF0000) >> 16);
    }
    else
        return (1); // must be a single-core processor
}
```

Determine maximum addressable cores

```cpp
unsigned int MaxcoreID_Package(void)
{
    unsigned int EAX=0;

    if (HTT_supported())
    {
        asm ("cpuid": "=a" (EAX)  
            : "a" (4), "c" (0)
            );
        return (((EAX & 0xFC000000) >> 26) +1);
    }
    else
        return (1); //must be a single-core processor
}
```

Determine initial APIC ID

```cpp
uint32_t InitialAPICID()
```
{  
  uint32_t EBX = 0;
  if (CPUIDsupported())
  {
    asm("cpuid": "=b" (EBX)
         : "a" (1)
        );
    return ((EBX & 0xFF000000) >> 24);
  }  
  else
  {return 0;
  }
}

// //////// Determine mask width in the initial APIC ID
unsigned int find_maskwidth(unsigned int count)
{
  unsigned int mask_width = 0, cnt = count;
  asm ("mov %1, % eax
          "mov $0, % ecx
          "mov % ecx, %0
          "dec % eax
          "bsr % eax, % cx
          "jz next%= 
          "inc % cx
          "mov % cx, %0
          "next%=:
          ": =r" (mask_width)
          ": r" (cnt));
  return mask_width;
}

// //////// Enumerate processor topology //////////
void Processor_topology(void)
{
  uint32_t APICID = 0, MaxLP = 0, Maxc = 0;
  unsigned int LPmask = 0, coremask = 0, SMT_ID = 0, Core_ID = 0, Package_ID = 0;
  APICID = InitialAPICID();
  MaxLP = MaxLogicalProcessor();
  Maxc = MaxcoreID_Package();
  coremask = find_maskwidth(Maxc); // mask width of the core ID
  LPmask = find_maskwidth((MaxLP / Maxc)); // Mask width of the SMT_ID
  SMT_ID = (APICID & ((1 << LPmask) - 1));
  Core_ID = ((APICID >> LPmask) & ((1 << coremask) - 1));
  Package_ID = (APICID >> (LPmask + coremask));
  if (! HTT_supported())
    std::cout << "This is a single-core processor and has no SMT_ID" << std::endl;
// //////// Read Model Specific Registers (MSRs) ////////

uint64_t _RDMSR(uint32_t MSR_addr)
{
  uint32_t EAX = 0, EDX = 0, greg[4] = {0, 0, 0, 0};
  uint64_t MSR_low = 0, MSR_64 = 0;
  _CPUID(1, 0, greg);
  if (greg[3] & 0x020)
  {
    asm volatile("rdmsr": "=a" (EAX), "=d" (EDX)
                  : "c" (MSR_addr)
                  );
    MSR_low = EDX;
    MSR_64 = (MSR_low << 32) + EAX;
    return MSR_64;
  }
  else
  {
    return 0;
  }
}

// //////// MP Configuration Table //////////

void MP_table(void)
{
  uint32_t i = 0x99cf0, addr = 0;
  while (i < 0xffffffff)
  {
    if (0x5F504D5F == *((uint32_t *)i))
    {
      break;
    }
    i += 4;
  }

  typedef struct MPFloatingpoint {
    uint32_t signature;
    uint32_t MPtable_addr;
    uint8_t Length;
    uint8_t Version;
    uint8_t FByte[5];
  } MPFloatingpoint;

  typedef struct mp_conf_table {
    uint32_t signature;
    uint16_t BaseLength;
    uint8_t Version;
    std::string OEM;
    std::string Product_ID;
    uint16_t OEMtable_size;
    uint32_t OEMtable_Point;
    uint16_t entry_count;
    uint32_t LAPIC_address;
    uint16_t EXTtable_length;
    } mp_conf_table;
MPFloatingpoint_table = {.signature = *((uint32_t*)i),
MPtable_addr = *((uint32_t*)(i+0x04)),
Length = *((uint8_t*)(i+0x08)),
Version = *((uint8_t*)(i+0x09)),
FByte[0] = *((uint8_t*)(i+0x0B)),
FByte[1] = *((uint8_t*)(i+0x0C));
addr = table.MPtable_addr;
if (table.signature == 0x5F504D5F && table.MPtable_addr != 0)
{
  mp_conf_table mptable = {.
    signature = *((uint32_t*)addr),
    BaseLength = *((uint16_t*)(addr+0x04)),
    Version = *((uint8_t*)(addr+0x06)),
    OEM = (char*)(addr+0x08),
    Product_ID = *((uint16_t*)(addr+0x0c)),
    OEMtable_size = *((uint32_t*)(addr+0x10)),
    OEMtable_Point = *((uint32_t*)(addr+0x14)),
    entry_count = *((uint16_t*)(addr+0x18)),
    LAPIC_address = *((uint32_t*)(addr+0x20)),
    EXTtable_length = *((uint16_t*)(addr+0x28));
}
}

#if // //////// ACPI tables /////////
void ACPI_table(void)
{
  uint32_t i = *((uint32_t*)0x040e0);
  while (1)
  {
    if (0x20445352 == *((uint32_t*)i))
      break;
    i += 4;
  }
typedef struct {
    std::string signature;
    std::string OEM_ID;
    uint8_t Rev;
    uint32_t RSDT_addr;
    uint32_t Length;
    uint64_t XSDT_addr;
  } RSDP_table;
typedef struct {
    uint32_t signature;
    uint32_t Length;
    std::string OEMID;
    std::string OEM_TID;
    uint32_t Entry_addr;
  } RSDT_table;
RSDP_table RSDP = {.signature = (char*)i, .OEM_ID = (char*)(i+0x09), .Rev = *((uint8_t*)(i+0x0f)), .RSDT_addr = *((uint32_t*)(i+0x10)), .Length = *((uint32_t*)(i+0x14)), .XSDT_addr = *((uint64_t*)(i+0x18)));
Listing A.2: Self-configuration code

```
org 0x800000
SECTION .text
BITS 16
GLOBAL _AP_start

_AP_start:
JMP boot_code
ALIGN 4

GDTPointer dd 0x80400
IDTPointer dd 0x80440
_COUNTER dd 0x80480
prime_addr equ 0x804c0
prime_input equ 0x804c8
_stack equ 0x300000
results equ 0x80500

ALIGN 4
boot_code:
   CLI ; disable interrupt
   CLD ; direction from lowest to highest address
   mov ax,cs ; reset the data registers in same address as code
   mov ds,ax
   mov ss,ax
   mov es,ax
   mov fs,ax
   mov gs,ax
   MOV ebx,[_Counter]
   lock INC WORD [bx]
   mov ebx,[GDTPointer] ; Loaded from bootstrap processor
```
lgdt [bx] ; load GDTPointer into GDT → register

mov ebx, [IDTPointer] ; Loaded from bootstrap ← processor
lidt [bx] ; load IDTPointer into IDT ← register

;;;;;;; Enable Protected Mode in APs ;;;;;;;;

mov edx, cr0 ; set bit 0 of processor ← control
or edx, 1 ; register CR0 to ← enable protected mode
mov cr0, edx

; Just small switch delay
nop
nop
nop

;;;;;;; Switch to kernel mode ;;;;;;;;

jmp DWORD 0x08:.Flush

BITS 32

.Flush:
  CLI
  CLD
  mov ax, 0x10
  mov ds, ax
  mov es, ax
  mov fs, ax
  mov gs, ax
  mov ss, ax

;;;;;;; Execute specific task ;;;;;;;;
  mov eax,1
cpuid
shr ebx,24
jmp extra
hlt

extra:
  mov edx,0
  mov eax, 4096
  mul ebx
  add eax, _stack
  mov ebp,eax
  mov esp,ebp
Listing A.3: Computation of nth prime number

```c
int prime (int nth)
{
    int counter=1;
    int y = 2;
    while(counter <= nth)
    {
        int x = 2;
        int prime = 1;
        while(x*x <= y)
        {
            if(y%x == 0)
            {
                prime = 0;
                break;
            }
            x ++;
        }
        if(prime)
        {
            counter ++;
            y ++;
        }
    }
    y--;
    return (y);
}
```
Appendix B

Scripts

Listing B.1: Run multiple single-core IncludeOS instances

```bash
#!/bin/bash
export _core=36
export vms=1
(( vms = $vms + 1 )) # Starts at IP=2

for nr in $(seq 2 $vms)
do
    vm="small$nr"
    mac=$(printf "%02X" $nr)
    file=IncludeOS_service.img
    /usr/bin/qemu-system-x86_64 --enable-kvm -drive file=$file,format=raw,if=ide
        -device virtio-net,netdev=net0,mac=c0:01:0a:00:00:$mac
        -netdev tap,id=net0,script=/root/IncludeOS_install/etc/qemu-ifup
        -name includeOS$nr -vga none -nographic -smp cores=\$_core,threads=1,sockets=1 -m 128 > /dev/null &
done

sleep 10
./test-MC-IncludeOS-UDP.sh
```

Listing B.2: Measure the time of the nth prime number computation

```bash
#!/bin/bash
prime=500000
vms=1
(( vms = $vms + 1 )) # Starts at IP=2
echo $(date) >> MC-timeUDP.txt
for ip in $(seq 2 $vms)
do
    echo $ip >> MC-timeUDP.txt
    sleep 10
    ./test-MC-IncludeOS-UDP.sh
    sleep 5
done
```

```
untime=$( TIMEFORMAT='%R'; time (python UDP.py $prime 10.0.0.$ip) 2>&1 >/dev/null)
  echo $ip/$prime with $_core VCPU $utime Seconds >> MC-timeUDP.txt
done

sleep 10

sudo ./killvms.sh
```

Listing B.3: Kill IncludeOS instances

```
#!/bin/bash
for pid in $(ps -ef | grep 'IncludeOS_service.img' | grep -v grep | awk '{print $2}');
  do
    sudo kill -9 $pid
  done
```