Operating system Evaluation for real time image processing on a multi-core architecture

Navin Goel

PHILIPS

TU Delft

Delft University of Technology

Operating system Evaluation for real time image processing on a multi-core architecture

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Navin Goel

4th June 2009

Author
Navin Goel

Title
Operating system Evaluation for real time image processing
on a multi-core architecture

MSc presentation
June 19, 2009

Graduation Committee

prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
ir. Patrick Bronneberg	Philips Healthcare
ir. dr. D. H. J. Epema	Delft University of Technology
dr. phil. H. G. Gross	Delft University of Technology

Abstract

Traditionally embedded applications were limited to dedicated hardware with limited functionalities controlled by a microcontroller. With the advancement of technology and the rising needs embedded applications now run on Consumer off the shelf (COTS) products. Using general-purpose operating systems makes the development cycle for these applications shorter by providing necessary hardware support and thus allowing the application to be developed on a higher abstraction. In this thesis, we develop a model to evaluate different operating systems for such an embedded application. The goal was to be able to predict the application performance on different operating systems without having to port or code the application for each operating system. The model uses micro-benchmarks on operating systems and then consolidates the results to give an overall score. The model thus developed was used to test and evaluate operating systems for a medical image processing application used in cardio vascular intervention procedures.

“Any fool can make things bigger, more complex, and more violent. It takes a touch of genius - and a lot of courage - to move in the opposite direction. ” –
Albert Einstein

Preface

This Thesis is a part of my Masters in Computer Engineering at TU Delft, Netherlands. This research was done at Philips Healthcare along with Parallel and Distributed Systems group, TU Delft.

I am more than grateful to the people and institutions that have made this possible. First and foremost I would like to thank my supervisor Patrick Bronneberg, who has guided, supported, inspired, and helped me throughout the project. Due to his great interest and insights in the project I have always been motivated during the past nine months. I am also greatly thankful to Prof. Henk Sips and Gernot Eggen for always taking out time to listen to the challenges faced by me and to try and look for various solutions and also for reviewing my thesis report. I would also like to thank colleagues at Philips Healthcare for enlightening me with all the discussions on various operating system related topics.

Surrounding it all I would like to thank my parents for always believing in me and for supporting and standing by me for my every decision, especially to come and study in Delft. Last but not the least I would like to thank my colleagues in Philips and my friends who constantly took me away from my work and kept me amused with movies, parties, food, darts, music, and their terrible sense of humour.

Navin Goel

Delft, The Netherlands
4th June 2009

Contents

Preface vii

1 Introduction 1

2 Background Study 5
- 2.1 Operating System 5
  - 2.1.1 Functionalities 6
- 2.2 Real time systems 13
  - 2.2.1 Dimensions of a real time system 13
  - 2.2.2 Real-Time Operating System for Multicores 14
- 2.3 Cardio Vascular System 17
  - 2.3.1 Use Case for angiography 18
  - 2.3.2 Parts of C/V System 18
  - 2.3.3 The Image-processing subsystem 20

3 Requirement Analysis 21
- 3.1 Image processing subsystem 21
- 3.2 Requirements 22
- 3.3 Prioritized List of OS functionalities 24

4 Testing Methodology 27
- 4.1 Background 27
- 4.2 Test Design 29

5 Tests, Results and Analysis 31
- 5.1 Networking 31
  - 5.1.1 CPU load test 32
  - 5.1.2 Jitter Test 33
- 5.2 Display Support 35

5.2.1 Latency Test 36
5.2.2 Frame time 37
5.2.3 Monitor Handling 38
5.2.4 Other Issues 39
5.3 Memory management 39
5.3.1 Available memory bandwidth 39
5.3.2 Memory bandwidth overheads 43
5.4 Device Drivers 44
5.5 Disk I/O / File system 46
5.5.1 Read / Write Bandwidth test 47
5.5.2 CPU overhead 52
5.6 Scheduler 52
5.6.1 Context switch time 53
5.7 Reliability 54

6 Consolidation Methodology **57**
6.1 The model 57
6.2 Performance Evaluation 60
6.3 Results 61
6.3.1 Memory Bandwidth 61
6.3.2 CPU usage 61
6.3.3 I/O 62
6.3.4 Application Overhead 63
6.4 Validation 64

7 Conclusions and Future Work **67**
7.1 Conclusions 67
7.2 Future Work 69
7.2.1 OS evaluation model 69

A Image Processing subsystem **75**

B Requirement Gathering **77**
B.1 Acquisition use case 77
B.2 Current system 81
B.3 Ideal operating system 82

C Future Work for IP pipeline **85**

Chapter 1

Introduction

This study was done within Philips Healthcare for the Cardio Vascular(C/V) system. The C/V system is used for diagnosing and treating problems in blood vessels in the human body. It aids the doctor to visualize blood vessels. Common problems are aneurysms and stenosis in the blood vessels. To visualize the arteries a highly complex set image processing system is used. The image-processing pipeline is the most important part of the C/V system since the system is primarily used for imaging. This thesis develops a methodology for evaluating the most appropriate operating system for the image processing subsystem.

Embedded applications are becoming more and more complex as are the underlying systems thus making it impractical to develop the full system from bottom-up. In practice, most of the embedded development is moving towards commercial off the shelf products as can be seen from the recent research in this area [14] [10] . Choosing a general-purpose operating system for an embedded application gives the advantage of having a good and continued support for a large range of hardware. This ensures that the embedded system can make use of the latest hardware developments while minimizing development time on the application for every hardware upgrade.

The right choice of operating system helps in gaining the best performance from the hardware. At the same time the choice of the operating system, also reflect on the amount of development time and effort required for an application. For example, an operating system, which supports real time application and takes care of scheduling requirements, will require less effort than an operating system lacking this support for a real time application. With another example for multi-cores, some schedulers run periodic inter-core load balancing algorithms, although useful for general purpose computing this can be a serious issue for specific ap-

plications. Similarly, some of the features of the operating system might be undesirable in some cases and thus might require extra effort to switch them off or to program against them. Since the choice of an operating system is a long-term investment, the right choice at the development time is crucial.

Embedded applications typically run exclusively on the targeted hardware and have real time requirements and or high throughput requirements [7] . Since every application has its own set of different requirements, different operating systems might be better suited for different applications. Apart from the functional requirements of the application, many non-functional attributes play an important role in the choice of the operating system like familiarity with the operating system, available development environments, costs etc. Thus, giving a ranking to the operating systems alone is not sufficient; we need a measure of applications functional performance relative to the different operating systems so that along with the non-functional attributes it gives a complete picture to judge the best-suited operating system for the application. This study gives the technical score to the operating system, which can then be used to compare along with the non-functional costs the overall operating system performance.

The ideal test for measuring an application's performance under different operating systems would be to program the application on each operating system and then to measure the actual performance. Having an application written on top of an abstraction layer that can run on different systems does not give a true reflection on its performance. On the other hand, coding or porting the application for each operating system is quite a tedious task, which costs a lot of time and effort. In addition, a ported application will not be able to utilize the optimizations that the operating systems offer completely. Since we cannot practically choose the best operating system for an application by directly running it, we need to do a comparative study of the operating systems.

We have developed a methodology to evaluate various operating systems for a particular application without having to port or code the application for each OS. This methodology is used to find the best-suited operating system for an image-processing pipeline used in cardio vascular system for interventional procedures.

To evaluate operating systems we propose an approach to predict relative application performance on various operating systems without actually having to program the application. We predicate that it is possible to combine the results of micro-benchmarks on the operating systems in accordance with the requirements of the application to evaluate the functional performance of the application on the operating system. We verify this hypothesis by testing the results against the IP

pipeline. This involved studying the different sub-functionalities of the operating system and benchmarking them in isolation. These individual results were then mapped to the IP pipeline requirements and coalesced to get a performance score for each operating system that was tested. Special focus was given to the real time and high performance requirements of the application on a multicore, multiprocessor system.

There are many benchmarks available to look at operating systems functionalities but these benchmarks are mainly focused on the hardware performance rather than on the operating system performance. Other studies only provide differences between the same class of operating systems [3] [12] . There are a few operating system comparisons but these are focused on only a particular aspect [5] or are too broad in comparison at the architecture level and not at the performance level [8] [20].

Four general-purpose operating systems were evaluated for the study. The operating systems Windows Client version 7, Windows server 2008 R2, Linux 2.6.28 (fedora) and Windows XP (used as a base) were used as they support a broad range of the available hardware choices.

In Chapter 2, the background literature study for the thesis is provided. We look at the structure and the various functionalities of the operating system, real time systems and the Cardio Vascular system, which is our application use case. In the next chapter, we will look at the application in more detail and gather our requirements from it.

In Chapter 4, we describe the testing methodology followed to develop various tests. Chapter 5 details the individual tests and there results followed by their analysis and the influence of those results on our application. In Chapter 6, the model for consolidating all the individual results from chapter 5 is given and the final score for each operating system in calculated. Finally, we end the thesis with conclusions and future work in Chapter 7.

Chapter 2

Background Study

This chapter presents the literature study done for the thesis and is divided into three sections the operating system, real time systems, and the Cardio Vascular system.

2.1 Operating System

An operating system is the piece of software that manages all the computers hardware resources and provides a layer on which the application runs. Modern operating systems apart from providing a layer between hardware and software also provide support for functionalities like multitasking, multi processing, and multithreading. For our further discussion, we consider an operating system to encompass everything in the system that is not part of the hardware or the user application and has an influence on the performance of the running application directly or indirectly.

There are different types of kernel architectures namely monolithic and microkernel [19]. A monolithic kernel is a kernel architecture where the entire operating system is run in kernel space as supervisor mode. The monolithic kernel defines alone a high-level virtual interface over computer hardware, with a set of primitives or system calls to implement all operating system services such as process management, concurrency, and memory management in one or more modules. The tight internal integration of components in a monolithic kernel allows the low-level features of the underlying system to be effectively utilized, making a good monolithic kernel highly efficient. In a monolithic kernel, all the systems such as the file system management run in an area called the kernel mode. All the modern consumer operating systems are monolithic with a modular structure having some

core services as a part of the tightly coupled kernel and other modules that can be dynamically loaded and unloaded.

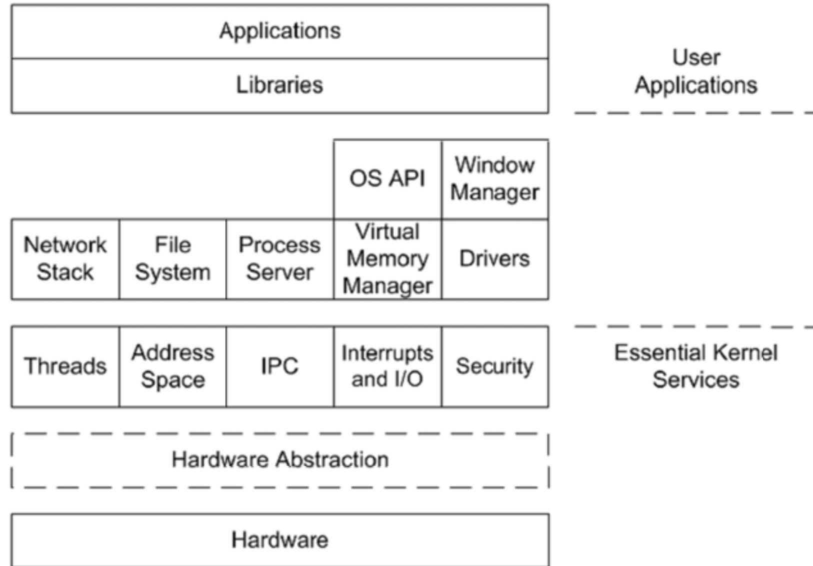

Figure 2.1: Structure of modern operating systems (monolithic kernel with modular structure)

To be able to evaluate an operating system for an application we first need to look at the various functionalities provided by the operating system to aid the application both directly and indirectly. The functionalities are broadly classified in to functional and non-functional aspects.

2.1.1 Functionalities

Device Drivers

One of the basic functionalities of the operating system is to provide an interface to the underlying hardware. This is done via a device driver. Device driver communicates with the device through the communication bus to which the device is connected. A device driver is hardware and operating system dependent. It is responsible to execute the high-level user commands from the application to the low-level instructions of the device in an efficient way. For an operating system

to make full use of the hardware, it is crucial that the hardware is well supported with drivers from the vendor. Especially in case of non-trivial hardware's like graphics cards, network interfaces, sound cards etc. Since device drivers run in kernel mode, it is crucial that they are reliable since an error will cause system hangs and crashes.

Scheduler

Scheduling is a key concept for multitasking operating system. A scheduler is responsible to enable multiple tasks to run at the same time on the same processor by enabling time based sharing of the limited resources. A scheduler typically has a list of all the tasks running on the system with their assigned priorities. It creates a run queue based on those priorities with the tasks that are ready to run.

An ideal scheduler would schedule the process with the highest priority in the next CPU clock cycle without wasting any clock cycles to assign the same. In practice, the scheduler also runs as a process that can manage all the processes in the run queue and assign the next one ready to run. The scheduler does this by assigning each process to a CPU for a time called the timeslice. The time slice should be small enough so that the wait time of the critical process when ready to run is the shortest. Since the scheduler itself takes time to run and consumes CPU cycles having a timeslice, which is too small, will mean excessive overhead by the scheduler. The time taken by the scheduler to schedule the next task is termed as scheduling latency. The complexity of which is measured with the number of tasks in the run queue. Many scheduling algorithms exist now, which are independent of the number of processes in the run queue and thus have a time complexity of O(1).

In real time systems a deadline for a critical task is very important and sometimes waiting for the timeslice to get over can be too crucial, thus, pre-emptive scheduling is used in this case. Pre-emption means stopping a task before its time slice is over to let a high priority task run. In a multiprocessor system, the scheduler is responsible to maintain the core affinity of the process to avoid extra latency due to cache misses. If a process is scheduled on a different core, the data needed by a process might have to be transferred from the cache of the old core to the current core of operation. The scheduler should be aware of this penalty and schedule the tasks accordingly.

Memory Management

Memory management provides a way to allocate portions of the main memory between different processes as needed and to free them to accommodate new processes. This task is critical to the efficient working of the computer system.

The main memory is divided into two levels mainly the primary memory and the secondary memory. Deciding which pages stay in the main memory and which pages stay in the secondary memory is a part of the memory management unit. Swapping a page to be used next can hamper the performance a lot as the delays of fetching a page can run into milliseconds.

The memory manager uses several algorithms to make sure that the memory pages are pre-fetched into the main memory before the actual execution of the process. It also takes care that a page is not discarded from the main memory that might be used in the future to prevent extra latency in running a process. Sharing of memory pages between processes (copy on write) to save space is also handled by the memory manager.

Networking

A network manager handles the input network I/O traffic to the system and analyses it to hand it over to the correct application or to discard it. It is responsible for implementing the network stack in the system and setting up the data flow.

With the increasing speed of network interfaces with the advent of gigabit and 10 gigabit Ethernet the CPU load on the system can be huge. If the processing of the network stack is not offloaded to the network card it also goes for the memory bandwidth if not handled properly. Figuratively speaking a TCP load of 5 Gb/sec translates to a CPU load of 5 GHz.

Disk I/O / File system

The file system is used to allocate space on the disk in a manner that allows the user to store files such that they can be retrieved back. The file system should provide support for data reliability and should work efficiently with both software and hardware based raid configuration.

Interrupt handler

Interrupt is an asynchronous signal from the hardware indicating that an action is ready to be performed. An interrupt causes a context switch in the system to kernel

mode, where the interrupt request is handled. Interrupts can also be software generated hinting the operating system about availability of some resource or expiration of a timer. A software interrupt normally indicates a change of state of the owning process to the scheduler and makes it ready for scheduling in the next cycle. An interrupt handler should handle interrupts with priority and with least delay possible. It should also be very cheap in terms of CPU cycles.

Resource Allocation

The operating system along with providing the abstraction layer to the hardware also manages the hardware resources between processes. The operating system is responsible for efficiently allocating and de-allocating resources to different processes. It also manages resource ownership and security. It is important that the operating system does this while considering the priority of the processes and ensures that a deadlock for resources does not occur.

Window Manager

This is the interaction window of the operating system, it can be either command line based or graphical. All modern operating systems support one or more graphical interfaces. The window manager by the operating system is responsible for showing the application (if graphical) to the user. For graphics intensive applications, this is an important part of the operating system as they rely on its effectiveness for their output when using the window manager to manage their output. Graphic applications have to share the graphics hardware with the window manager when managing the rendering themselves. The rendering API's are also dependent on the window manager.

Security

A computer's security depends on a number of technologies working properly. A modern operating system provides access to a number of resources, which are available to software running on the system, and to external devices like networks via the kernel.

The operating system should be able to differentiate between request that are safe to be processed and the ones that can hamper the system and thus stop their execution. In addition to allowing and blocking of various actions, the system must allow options to audit various resources and system settings.

The manufacturer should release security fixes to known vulnerabilities periodically. The system should be updated automatically but in no way should the security updates hinder the working of the application or change the way a user interacts with it by any changes made to the kernel.

Threading

Multithreading allows multiple threads to exist within the context of a single process. These threads share the process resources but are able to execute independently. This allows a single process to execute parallelly on a multiprocessor system. It allows fast switching of context between different threads. A thread is usually cheaper than a process and allows the user to do a fast context switch between threads of the same process. The operating system may or may not offer support for thread based scheduling. In the later case, the scheduler relies on the thread to relinquish control to stop executing.

Backward compatibility

The operating system should support any old hardware or software required by the application and the new versions of the operating system should also support the application.

Scalability

The operating system should be able to scale if needed to support more processors or memory for example.

Proprietary Technologies

Although not a function of an operating system but some technologies are tied with an operating system for example DirectX with Windows, various file systems, security protocols etc. If an application relies on one or more technologies that are native to an operating system, it becomes important to evaluate the cost of switching the technologies from one OS to another.

Interrupt service routine

An interrupt service routine (ISR), is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt. An

ISR is initiated by either hardware or software interrupts, and are used to service hardware devices or critical software operations like system calls.

An ISR causes a jitter in the current process execution in the system and thus should have a very short execution time. An ISR blocks all other processes and interrupts of equal or lower priority until its execution is finished.

Inter process communication

IPC in the operating system enables exchange of data between different processes or threads running on the system.

Reliability

Reliability is the most important non-functional aspect of the operating system. The operating system should be predictable and robust.

The operating system should be able to shield the system against any crashes and or data losses. This usually translates to having longer uptimes. It is not possible to prevent system crashes fully for any system. However, in case of a system crash, the application should not lose data and should be able to restore itself after a reboot and the reason for the fault should be easily diagnosable.

The system should also be predictable in the results produced with the same environment. This means that if an application runs with a specific system load and achieves a given performance it should be able to give the same performance each time it is run under the same load.

The faults are not only software generated but can also be hardware generated, not all hardware faults should be fatal to the system. The operating system should be able to cope up with minor hardware faults like bad sectors on the disk or a bad USB device plugged into the system.

OS-overhead

OS-overhead is the quantified cost of running the operating system also termed as kernel footprint. In a perfect world, the operating system would operate without using up any resources but that is impossible. Since the cost of running the operating system is an infrastructural overhead it should be as low as possible allowing the applications on top it to be able to gain the benefit of full hardware utilization. Since kernel is more or less resident in the system at all times having a componentized kernel gives the option to throw non-used components out reducing its footprint.

SDK

Having a perfect environment for running the application alone is not enough, the operating system should also provide proper tooling and programming support for the developer to be able to build and debug his application with ease.

Installation - software packaging

This refers to the distribution of the application and its installation. The operating system environment should allow for packaging the software with dependencies as far as possible and it should help the application identify the missing dependencies at the time of installation and to allow necessary actions for the same. The operating system environment should also take care of any additional privileges that might be required by the application and verify the same against the user rights.

Maintenance - remote administration

After the development and deployment of the application, it is sometimes necessary to do some updates or fix some bugs that were not discovered in time. In addition, it is cheaper to resolve some particular issues remotely. All this requires the support of remote maintenance and administration. The operating systems should provide support for this.

Cost of the OS

The cost price of the operating system and the professional support are also deciding factors for the OS.

Portability

Portability allows being able to run the application on a different environment without the need to create new code.

Support

There should be professional support available for the operating system in consideration which can help solve bugs and clarify issues that might be encountered during the development process.

2.2 Real time systems

According to Halang [9] ”Real time system is defined as the operating mode of a computer system in which the programs for the processing of data arriving form the outside are permanently ready, so that their results will be available within predetermined periods of time; the arrival times of the data can be randomly distributed or be already a priori determined depending on the different applications”.

2.2.1 Dimensions of a real time system

To define the complexity of a real time system there are many dimensions that are taken into consideration[18]. Let us look at these dimensions briefly.

Strictness of the deadline

Real time systems are categorized into two classes namely hard real time and soft real time. The hard real time systems are those in which completion of a task after its deadline is considered useless and can lead to complete system failure, in other words all the deadlines should be met on time e.g. a flight control system where a deadline miss could be fatal. For a soft real time system, the deadlines are more relaxed and if one deadline is missed, it is not fatal like in case of video streaming, although the frequency of misses has to be quite low.

Granularity of the deadline

This is the second factor while looking at the real time system. The granularity of the system can run from a few nanoseconds up to 100’s of milliseconds. With the granularity, laxity also plays an important role. Laxity is defined as the tightness of the deadline, i.e. a measure of the amount of work to be done in the period allocated. The type of environment selected for the application depends a lot on the granularity and laxity of the deadlines.

Size of the system

In most real time systems, the application size is quite small enabling the whole system to be loaded into memory. This simplifies many aspects of building the real time system but as we go towards systems that are more complex, the size increases manifolds and it is no longer feasible to assume that the whole code can fit into the main memory and memory management becomes important.

Degree of interaction of the system

If each subsystem of the application is mostly independent of the other subsystems then it can be run with limited co-operation among the tasks and its predictability quotient is contained within the application itself. However, as the subsystem becomes interdependent the predictability of the application is more complex to guarantee and new problems arise that have to be taken into consideration. The environment might also force interdependence on the subcomponents like with limited hardware that has to be shared between subcomponents.

Environment

The real time applications can sometimes run in a static environment where all the variables are controlled and the environment is 100% predictable, for example in an ASIC or in a microcontroller with the RT application running directly on it. However, more often than not the applications are run in dynamic environments, which are complex and then predictability is not a given factor any more for the environment for example a RT application running as a process on an operating system. It is claimed that having a system where the hardware will never fail and the software is bug free is impossible; but the real time environment must still be largely predictable and should be available at an acceptable cost.

Considering all the above factors the requirements of a real time application can vary from one application to the other. In our further discussions, we will focus on the operating system as the environment for our real time application on multiprocessor architecture.

2.2.2 Real-Time Operating System for Multicores

A real time operating system requires having a predictable behaviour. In theory, the time taken by any kernel modularity should be deducible to a mathematical formula giving precise notion of time. Non-predictability of an operating system can induce delays in the software pipeline that eventually lead to deadline misses. A general-purpose operating system on the other hand is not built keeping real time requirements in mind. The commercial operating systems available in market like Windows and Linux provide some support for soft real time applications and other specialized operating systems such as VxWorks or Windows CE provide support for hard real time systems. The lack of support for hard real time applications in Windows and mainstream Linux does not imply that no hard real time applications can run on those platforms. The choice of operating system (RTOS/ general

purpose OS) depends on each application's real time dimensions.

In this section, we will study what are the additional functionalities that an operating system can provide to facilitate running of real time applications. We will also study the characteristics that are specific to real time operating systems on a multiprocessor architecture. In a research by Stankovic [16], he reflects that real time task and scheduling problems tend to become worse as more hardware (multiprocessor) is used thus requiring additional understanding of the architecture by the operating system.

Task Scheduling

Scheduler is one of the crucial subcomponents of a real time kernel. A real time scheduler uses a priority based pre-emptive algorithm to schedule tasks. A preemptive scheduler stops any running task whenever a higher priority based task is ready for execution. The scheduler should take care of priority inversion, a phenomenon that occurs when a resource required by a high priority task is held by a low priority task blocking the execution of the high priority task and a middle priority task comes in thus taking precedence over both the low and the high priority task. An event that hampered the operation of mars pathfinder [2].

When a new task is ready to be run; either due to a higher priority task being ready or when the current tasks time slice is over the system invokes the scheduler. This is followed by a context switch from user mode to kernel mode and then a task is picked form the list of ready tasks, which should be run next. Next is to save the system state for the current task load the new task into the system and then make another context switch back to the user mode. For the scheduler to have a deterministic behaviour the above steps need to be finished in constant time. That means that the scheduler algorithm should be O(1) in complexity (i.e. takes constant time with reference to the number of processes to be scheduled) and should be fast that means that the cost of both scheduling and the context switch time should be minimum [17]. The scheduler should make a clear distinction between real-time tasks and non-real time tasks and then allocate the processors accordingly.

In a multi processor architecture the scheduler should try to schedule the task nearest to the core where its data is located if it is not possible to schedule on the same core, this requires the scheduler to have knowledge of the architecture and NUMA support from the hardware. Having NUMA support in hardware enables the processor to know the latency cost involved in accessing different memory banks. This knowledge can be used to make intelligent decisions on where to allocate memory for a new process.

Predictability

Real time systems are quite often mistaken to be synonymous with fast systems; which is not always true. The major difference between a fast system and a real time system is the need for predictability of the system. Even the fastest of the systems cannot guarantee that tasks will finish in a certain strict time deadline 100% of the time. Predictability and not speed is the foremost goal of any real time system.

In complex environments, many different aspects have to be determined to predict the system behaviour. For a deterministic operating system it is required that the worst-case execution time for all its system calls is calculable. For an operating system to be able to support hard real time systems it is necessary to have guaranteed worst case interrupt latency and context switch times.

Timing resolution

The operating system should have support for real time clock that has a resolution in accordance with the granularity of the real time application. It should also provide for special alarms and timeouts to ensure proper running of the application. Tasks should be able to hand over the control back to the operating system for limited short idle periods of time to ensure working of the operating system and or for other applications to run at the same time ensuring that the deadlines are met.

Context Switch Time

A context switch is the computing process of storing and restoring the state of a CPU such that multiple processes can share a single CPU resource. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. Each time a process is ready to be scheduled the operating system has to do a context switch to it. This is the overhead of doing multiprocessing on a system. The context switch is also dependent upon the granularity of the application if the task takes 100us and so does the context switch that will mean only 50% throughput from the system.

Interrupts

Real time applications require a guarantee on the interrupt response time with its granularity varying on the application. Interrupt latency is the total length of time

from an interrupt signal arriving at the processor to the start of the associated interrupt service routine (ISR). When an interrupt occurs, the processor must take several steps before executing the ISR. First, the processor must finish executing the current instruction and identify the interrupt type. This is done by the hardware and does not slow or suspend the running task. Finally, and only if interrupts are set, the CPU's context is saved and the ISR associated with the interrupt is started. In addition, interrupts might be blocked for critical sections of the code and that adds to the total delay an interrupt might have to wait before it is addressed. For multiprocessor system, support for intercore interrupts is a plus.

Componentized Kernel

Having a kernel that has the option to dynamically add and remove components can aid the developer in removing the non-required parts while keeping the operating system scalable for future needs. Having a small kernel guarantees that the interrupts and context switches in the operating system are low. This ensures higher predictability as the system becomes less dynamic. On the other hand, configurability allows the operating system to accommodate any hardware changes in the system or any software needs that may rise with time.

Communication

The real-time communication subsystem should be able to predict and satisfy individual message-level timing requirements. Interaction is an important aspect for a real time system. The communication can range from inter process communication to network communication. Each of these has different requirements from the operating system. For e.g. for a predictable network communication, we need intelligent 'network buffer management' support from the environment along with network scheduling which is coherent with the processor scheduling.

2.3 Cardio Vascular System

Cardio vascular X-ray systems are designed for interventional cardiology, electrophysiology and neurovascular procedures. It is used to prevent and cure chronic diseases which are the number one ranked cause of death [1].

Interventional cardiology is a branch of the medical specialty of cardiology that deals specifically with the catheter-based treatment of structural heart diseases.

This most commonly involves the insertion of a sheath into the femoral artery and cannulating the heart under X-ray fluoroscopy, real-time x-ray visualization.

2.3.1 Use Case for angiography

Angiography is used by the physician to look for stenosis and or aneurysms in the blood vessels. The physician first inserts a catheter into the femoral artery which is guided to the affected area in the heart via the help of x-ray scans taken when necessary by the use of the fluoroscopy. Fluoroscopy is a low dose X-ray procedure given to the patient to get real time moving images of the internal structures of a patient. When the catheter is placed in the area that the physician wants to visualize the physician makes an acquisition of the area. This is done with the help of fully rotating arms equipped with X-ray generator and detector which can be placed in any direction to get a good view of the area of interest. Many different acquisition scenarios are pre-configured in the CV system which allows the physician to get the images with the press of a single button.

The blood vessels are transparent to the X-ray thus a high contrast agent is added to the blood which can then absorb the x-ray showing the blood vessels in the image. A lot of different algorithms are applied on top of the standard noise reduction algorithms. This gives the physician a clear view of the heart's arterial venous system; this is used to look for abnormalities like aneurysms or blockages. Then the X-ray arms might be moved by the physician to get a better view on how to navigate to the suspected blockage or aneurysm and to also allow looking around obstacles like other arteries in front or maybe a pacemaker. After the acquisition the physician decides on the best way to reach the blockage or aneurysm to be able to fix it by using an inflatable balloon or with the help of a stent. From the acquisition a 3-D model of area can also be generated to ease the physician in creating a navigation roadmap or to aid in quantitative analysis.

The X-ray dosages are kept to a minimum to avoid exposing the patient to a lot of radiation, which means increase in the image noise and a lot more image processing has to be done to have better final image quality. The acquisition can be replayed by the physician to review and study to decide upon the procedure.

2.3.2 Parts of C/V System

The CV system is placed in two rooms, the exam room and the control room. The exam room is used by the physician while operating on the patient and has the controls for the X-ray, geometry and the image processing. The control room

is used for general workflow and for reviewing the acquisition images and has the control for reviewing. The exam room has the exam monitor to view the live acquisition images and also reference monitors, where as the control room has the view monitor for reviewing and data monitor to view the clinical data about the patient.

Figure 2.2: Cardio Vascular System

The CV system functionalities can be roughly broken into the following

1. Positioning subsystem

   The CV system has a lot of movable parts which allow the physician to scan all the areas of the patient from various angles. The movable X-ray arms can be positioned oven any body part with precision and the arcs are very stable while taking the x-ray to avoid any noise due to the movement of the x-ray. The arms can also take a fully rotational 3-D scan of the area of interest. The movable parts have collision detection system which prevents the arms to bump into the patient, the physician or the patient table while moving. The positioning system can be controlled by the physician with the help of the user panel provided. The positioning subsystem is a real time system as it has to take care not to harm the patient or the physicians.

2. IP subsystem

   The IP subsystem takes care of all the image processing. It gets the raw images as input from the flat panel detector and is responsible for processing those images to reduce noise and apply various algorithms to give a better view to the physician showing the images on up to 8 screens. In the acquisition phase the physician relies on these images to spot any abnormalities and to guide the catheter therefore the latency of the IP subsystem has to be short enough to allow hand eye co-ordination of the physician. The IP subsystem is also real time and needs to be free from any gliches or jitters.

3. X-ray generation and detection

   The X-ray generators are placed in the movable arms of the system to allow full body scan and flat panel detectors are used to catch the x-ray images.

4. User I/O handling

   The interface to the CV system and all the subcomponents is provided via special user friendly input modules.

2.3.3 The Image-processing subsystem

Image processing is a core part of the cardiovascular x-ray system. The image processing subsystem within the cardiovascular x-ray is responsible for receiving the input images from the x-ray detector(s), input commands and or parameters; it internally allows storing and manipulating of the images which are then displayed as the output along with generated information. It gets the input images from the x-ray detector which has high noise (low x-ray dosage) preventing the risks due to excessive radiation. The images are then enhanced and passed through various filters to remove noise and unwanted artefacts. As this subsystem is used in a critical system high degree of reliability is required. There are two main variants of the system single channel and the bi-plane configuration, the single channel has one x-ray arm where as there are two x-ray arms in the biplane configuration.

The Image processing subsystem is described in more detail in the Appendix A

Chapter 3

Requirement Analysis

3.1 Image processing subsystem

The IP subsystem runs as an embedded system within the CV system. The IP pc is a headless pc with no direct user interaction or input devices connected to it. It is responsible for processing data coming from the x-ray detector and displaying the processed images on the displays. The IP pc is also responsible for storing the acquired images, which can be processed and replayed later. We analyze the basic requirements of the system based on the current system but keeping in mind that the requirements for the system will increase for future versions.

There are two main use cases for the IP pc, Acquisition and Review.

The Acquisition is a live operation use case; there are tight upper bounds on the total latency delay of the whole CV system. The acquisition is used by a cardiologist to examine the blood vessels to look for any abnormalities and to guide a catheter through the arteries in the patient's body mainly in cardiac and vascular regions. The cardiologist is completely dependent on the images shown to him to perform the procedure. The images thus have to be processed in real time and without any jitter and the complete latency should be such that the physician can maintain his hand-eye co-ordination. One of the goals of the image-processing subsystem is to be able to work with high noise images efficiently. This allows the physician to be able to work on the patient with low radiation exposure.

In the review use case, the images stored in the system during the acquisition phase can be replayed by applying different imaging algorithms and parameters to enable the physician to get a good overview of the patient without having to expose him/her to radiation again. The acquisition stored are also used as a record for the patients history and sometimes by other doctors as a reference.

The image-processing pipeline is thus a real time high processing application. The real time dimensions of the application are summarized below:

- Strictness: - The real time deadlines are strict in nature and missing one deadline is considered fatal for the system. Although, each deadline is spread over a few milliseconds (the VSync) time and therefore the application can handle some jitter that is less than half the VSync time (8ms for 60 Hz).
- Granularity: - Although the granularity is low with the latency of the system running in milliseconds, the laxity is quite high with several billion operations to be performed within the period.
- Size: - The size of the system is kept in such a way that it can be fitted completely into the main memory. Each of the subcomponents is assigned a memory budget, which ensures that all the processes can fit into the memory.
- Degree of interaction: - The system is not only dependent on the other processes within the same pc but also on external (network) interfaces to provide the data flawlessly and thus has a high inter-dependence between the components.
- Environment: - The environment of the application is chosen to be an operating system, which is dynamic.

To analyze the working of the IP pipeline we will looked at acquisition use case, as it is more critical of the two and evaluated its requirements from that viewpoint. The detailed Requirement Analysis is given in the appendix B

3.2 Requirements

From the requirement analysis, we can summarize the OS requirements of the IP subsystem. In addition, we can categorize the requirements of the IP pc into real time requirements and high performance requirements.

Real time requirements

- Low network stream jitters
  - Upper bound context switch time
  - Constant network ISR time

  - Constant network stack processing time

- Scheduling
  - Predictable behaviour
  - Pre-emptive scheduling
  - Low jitter in scheduling latency
  - Maintain core coherence
- Fixed inter-process communication costs
- Constant frame upload times
- Predictable OS behaviour

High performance requirements

- Low context switch cost
- Low CPU load from network streams
- In-stream message processing
- Low scheduling latency
- Fast Disk I/O with multiple threads of access
- Efficient memory bandwidth management
- Low Network load
- Low cost in displaying image to the monitor
- Low OS overhead

Reliability

- Data reliability
- High uptime
- Low number of known and unknown bugs in the system
- Supportability

Management

- Ease of remote maintenance
- Low Costs
- High security
- Good SDK availability

3.3 Prioritized List of OS functionalities

The list of functionalities of the operating system for the IP pc was analyzed and a prioritized list was made from the above requirements. Each functionality was given a weight according to the extent in which it influenced our application. The priorities of the requirements were then discussed with the software development team currently working on the system. The current system bottlenecks were also taken into consideration to prioritize the requirements. Then the two lists thus obtained one from our requirements analysis and the other from the software development team were merged and a combined list was formulated. The list with their calculated weights is given in the table 3.1. The operating systems were tested against the first 8 functionalities.

Table 3.1: Prioritized list of functionalities with their weights

Group	Function	Score
I/O	Networking	27
Graphics	Display Support	27
Memory Management	Memory Management	26
Kernel	Predictability	26
Hardware	Device Drivers	24
I/O	Disk Access / File System	22
Scheduling	Scheduler	18
Management	Reliability	17
Kernel	OS overhead - Kernel time	15
Threading	Threading	15
Security	Security	15
Management	Maintenance	14
Memory Management	Resource allocation	14
Communication	Inter process communication	12
Management	Support	11
Management	Licensing / Costs	11
Manufacturability	Installation	11
Kernel	Scalability	10
Hardware	Interrupts	8
Management	Portability	8
Scheduling	Interrupt Service Routine	7
Management	Backward Compatibility	7
Development	SDK Compatibility	6
Kernel	Componentized Kernel	5

Chapter 4

Testing Methodology

4.1 Background

The perfect test for measuring an application's performance under different operating systems would be to run the application on each operating system and then to measure the actual performance. This would make choosing the best operating system for that application a straightforward task.

Writing an application that can be run on different operating systems without much work involved for porting will not give us a clear picture of the results. A real time application is hard linked to the operating system it is running on, this ensures real time guarantees and high performance by removing intermediate layers of abstraction. The application should exploit the knowledge of underlying operating system behaviour. This ensures that the application would meet its real time requirements and still yield high performance. Thus having an application written on top of an abstraction layer to ensure operating system independence will not give a true reflection on its performance.

On the other hand, coding or porting the application for each operating system is quite a tedious task. This would require a deep study of all the different operating systems, coding environments, debugging tools etc., which would cost a lot of time and effort. Thus, we can conclude that it is not feasible to measure the performance of an application on various operating systems by having direct application performance measurements.

Since we cannot practically choose the best operating system for an application by directly running it, we will have to compare the operating systems at a finer level. We do the comparison by benchmarking the performance of operating system's various sublevel functionalities that are crucial to the application. We will

first do an analysis of the application requirements noting the various roles the operating system plays in its execution. Then we can do benchmarking of those functionalities of the operating system that have an effect on the performance of the application. Afterwards we will have to combine all the individual results hence obtained to give a coalesced result of the performance numbers.

The term performance used here does not only refer to the throughput achieved but also encompasses the quality and predictability factors. Only by taking into account all the functional and non-functional requirements can we make an informed decision for the choice of operating system. The tests for IP pipeline were mainly focused on the speed and the real time guarantees of the system.

With the current complexity of the operating systems, it is not realistic to judge its performance by just looking at their theoretical specifications or by having a generalized performance benchmark. The question of the best operating system is also redundant if not asked in the context of an application. We need to develop targeted tests (micro-benchmarks) to look at the performance of the various (sub) components. The tests have to be designed such that they allow us to look at the performance of the individual functionalities of different operating systems in an unbiased fashion.

After we test the various sub-components of the operating system that are required by the targeted application, we have to combine the results obtained to a performance measure. This will give a clear indication on which operating system is better choice for the application under consideration. To combine the results we need to understand the importance and the extent of the role of every function to be able to provide it with a weight. We will study and analyze the consolidation methodology in depth in chapter 6 .

Due to the high complexity of the operating system, looking at it from a higher level does not reveal the true intricacies of the system. An application has several points of interaction with the operating system, all of which have incoherent effect on the overall performance. The operating systems subcomponents are highly interconnected and thus they have an influence over each other's performance. However, the effect on the performance of one (sub) component is affect in different ways in different scenarios from another (sub) component. This makes it important that the benchmarking is done by making sure that the targeted (sub) component is independent of influences from other (sub) components as much as possible.

The system hardware plays the biggest role in determining the performance of a system, needless to say that the benchmarks are dependent of the underlying hardware. Thus, all the operating systems were evaluated on the same hardware.

In addition, the operating system was found to have no noticeable effect on the pure hardware performance numbers like CPU clock, floating point operations per second, maximum network throughput, seek time for hard disks, hardware write time on HDD, inter core bandwidth, cache access times and memory latencies

Another reason for having the right choice of operating system is to gain that extra performance, which is no longer possible to gain by adding more processing power in the hardware. In fact, in many cases adding more hardware no longer improves performance, the advances in hardware have been tremendous and now it is at a point where the hardware no longer scales linearly and to get more performance improvement solely through more advanced hardware is quite costly. Hence, most of the work to gain performance is being done in improving software algorithms and environment efficiency. The correct choice of operating system is a crucial and a cost effective measure in gaining the best performance from the hardware for the application.

4.2 Test Design

A deep understanding of the operation of the operating systems functionality is required before a test can be developed for that specific (sub) component. To develop the tests for a (sub) component a thorough understanding of the role and working of it was gained through available literature and the study of the operating system. This was helped in deciding which performance measurements are required for the functionality and how can we measure those.

The tests were then developed by a recursive approach wherein first a test was conducted with the knowledge from the literature survey done, then after observing the behaviour and matching it with the theoretical expected results it was further refined to the requirements. The same test was then done on a different operating system and the same cycle was repeated. This was done until a common test for all the operating systems could be developed and it was independent from the influence of other (sub) components. Since there can be different ways the operating system handles the same request it is important to take care that the tests measure the same performance measure on different operating systems.

The tests were judged based on the results obtained and the differences from the theoretical expected values. The tests were improved with the help available from the experts in Philips and or in some cases contacting people working in Intel.

The operating systems evaluated were:

- Linux 2.6.28 x64 - FC 10

- Windows 7 x64
- Windows Server 2008 R2 x64
- Windows XP x32 (Reference)

Windows XP is currently being used in the current IP system development and was thus taken as a reference to measure the performance of the other operating systems. In addition, Windows XP was fully optimized for the IP pipeline application whereas the other operating systems were used out of the box. Windows 7 and Windows Server used were pre-release versions and thus there is a possibility that some of the results might change after the final release.

Most of the literature and tools available for performance measurement focus on hardware benchmarks. The few literature which do compare some functionality between different operating systems are focused among the same flavours of operating system like windows family or *nix which is relatively easier given the common kernel structure and thus ease of testing in the same way for all the operating systems in consideration.

In case of windows operating system being closed source it was difficult to understand the internal working of the components. A lot of effort was done to try to understand the working of the operating system as a black box. It was thus difficult to foresee some of the deeper functionalities in isolation. On the other hand, the workings of the functionalities that are open to the public are quite well documented and the literature available for those was very useful. Most of the literature was found via the MSDN portal and by consulting Microsoft directly.

Finding an up-to-date documentation for Linux subcomponents was a tedious task. Most of the documentation was outdated due to the dynamic development of the kernel or it was poorly written. However, being open source you could always look into the code for a deeper understanding of the internals. For the latest Linux changes, LWN was quite helpful along with various internet mailing lists.

A number of operating system optimizations can be applied to the operating system to improve performance of one (sub) component at the cost of another. Taking all the optimizations suited for an application into consideration was out of the scope of this study in the limited period and thus we use the operating systems directly out of the box except for XP that has been optimized for the current IP system and thus is used as a reference.

Chapter 5

Tests, Results and Analysis

This chapter presents the development of the micro-benchmarks and the results hence obtained for each operating system. We will also look into the results and try and figure out the reasons for the behaviour observed and then discuss what those results mean for our application.

5.1 Networking

Networking subcomponent is responsible for handling all the incoming and outgoing network data traffic in an operating system. While sending it takes care of everything from the point the application hands over the data, protocol information and the target address to the operating system till the point the packets leave the NIC (network interface card). Similarly, for the incoming traffic it takes the packets handed over by the NIC, processes them and finally gives data to the application.

We will look at the process to handle network packets in the operating system with more detail to be able to understand and benchmark it. The data is first broken into packets and then into IP frames to be able to send over the network. When a frame arrives at the network card, a hardware interrupt is generated by the NIC to signal the operating system of this event. The operating system then handles the interrupt by invoking the Interrupt Service Routine for the network interface; this ISR is provided by the network card drivers. The frames are assembled to form a packet, which is processed by the network stack in the operating system. The packet is checked for transmission errors against its checksum and then it's header is processed and finally the data is handed over to the targeted application. When the ISR is called, the processor does a context switch to save the current state of the processor and after the ISR is handled another context switch back the previous

running application is made.

This is a general summary of the events happening; there can be alterations that a system may utilize to handle the packets in its own way. Network cards can offer hardware offloading of the segmentation and checksum. In addition, many cards offer Interrupt Coalesce, which only interrupts the operating system when there are a specific number of packets in the queue or when a timer is expired.

To evaluate the networking component in an operating system we need to look at the CPU utilization for both transmitting and receiving data and the maximum throughput. We found that the maximum throughput was hardware limited and not hampered by the operating system used. We will also look at the memory bandwidth utilization by the networking subcomponent when we look at the memory management. Along with these performance attributes, we also look at the jitter in processing the packets. Since the IP stack and the interrupt service routine are inherent part of the networking subcomponent, we will not differentiate between them and measure the performance as of the networking subcomponent as a whole.

5.1.1 CPU load test

To measure the CPU load due to the networking subcomponent a small socket-networking program was written which could send and receive UDP packets. The program generated 8 KB UDP packets that were sent to another identical system connected directly over via a 1 GBps network. The program generated full network load on the system both wile sending and receiving data. The CPU load was then monitored while there was nothing else running on the system. Since the program itself did not have any significant computing involved and was consistent on all the operating systems, we can take the CPU load to be purely due to networking subcomponent.

Results

Table 5.1: CPU usage for receiving and sending network data

OS	CPU load (receive) %	CPU load (send) %
Linux	0.7	2.7
Windows 7	3	8
Windows Server	2	6
Windows XP	6	7.5

From the results in table 5.1, we can see that Linux outperforms other operating systems considerably in terms of CPU load. Windows 7 and server are next in line and Windows XP behaves the worst of all.

A generally accepted rule of thumb is that 1bps of network link requires 1Hz of CPU processing and as validated by [6] this rule more or less still holds. In our case, we had a network load of 1Gbps, which is equivalent to processing power of 1GHz. This translates to a little less than 6% CPU load in our setup. From the result, we can conclude that for sending Linux takes 0.5Hz per bps of network traffic whereas Windows server takes exactly 1Hz and Windows XP and 7 take a bit higher. This restates that the law is still valid for Windows operating systems whereas it has halved for Linux at full loads. These results reflect on the implementation of the network stack on the various systems. We know that for windows XP the additional receive load is due to the lack of support for de-fragmentation offloading to the NIC.

5.1.2 Jitter Test

To measure the jitter in the processing of the packets we have to look at the time at which the first frame for a packet arrives at the network card and the time that the data for that packet is handed to the application. Since the packet, sending was done at regular intervals without any jitter and same for the data transfer at full load we have a regular arrival pattern of the packets. Therefore, we can just look at the difference in the arrival time of two packets and that would give us the jitter in processing.

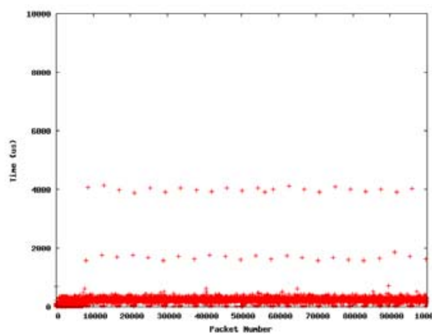

Figure 5.1: Windows 7 full plot

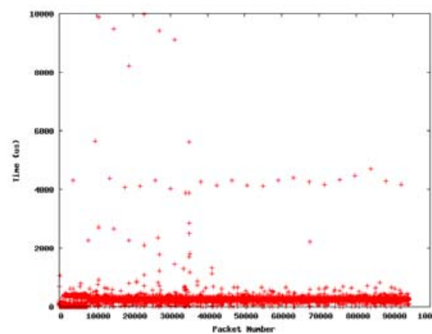

Figure 5.2: Windows Server full plot

To look at the timing for the jitter the program from the CPU load test was modified to write, the receive timestamp of the packets to a file so that the jitter

in the processing delay could be measured. The time stamps were recorded as the time difference between the arrival time of two immediate packets. The program was run for 100,000 packets and the difference in the arrival time was then plotted as a graph to analyze the jitter.

From figure 5.1 and figure 5.2 we see that most of the values are relatively very smaller than the few peaks we have. In both windows 7 and server 2008 there is a peak of 4 ms that occurs once every second and an additional peak of 2 ms every second that occurs in windows 7. This behaviour was not observed in the other two operating systems (Windows XP and Linux). Windows used deferred procedure calls to handle interrupt requests, when the system is already handling an interrupt the other interrupts are queued. It was found that in the Windows 7 and server 2008 R2 there was a delay of 4ms and 2ms every second. This can be due to an interrupt that is triggered every second and takes 4ms to complete. Now we plotted the graph by zooming to the area where most of the packets lie (>99.9%). We can now compare all the operating systems by looking at the jitter.

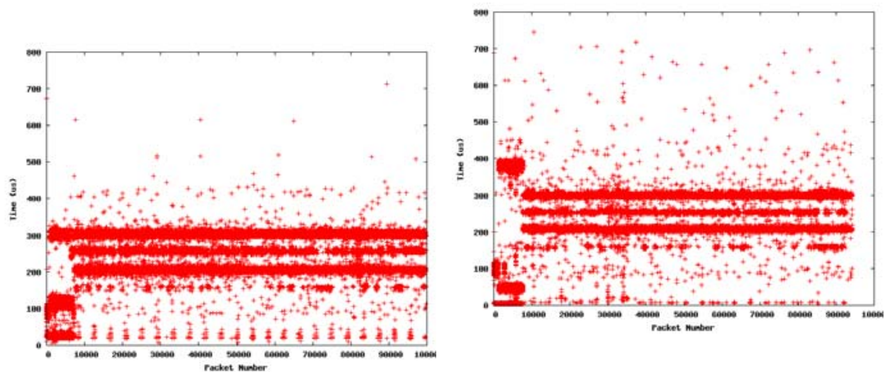

Figure 5.3: Windows 7 zoomed plot

Figure 5.4: Windows Server zoomed plot

By taking into considerations only the values, that we see here, we can see that the packets arrive in three different bands. The bands are clearly embarked for Linux (Figure 5.5) , Windows 7 (Figure 5.3)and Windows server 2008 R2 (Figure 5.4). The overall jittering is least in Linux followed by Win Server 2008 R2 and then Windows 7 whereas Windows XP has the worst spread. The bands are accounted for due to the 'Interrupt Coalesce' in the NIC, which collects a bunch of packets (3 in our case) before interrupting the OS . Interrupt coalesce waits for a number of packets to arrive before handing them over to the operating system to reduce the number of context switches [15]. Since the application has to wait for a number of packets before it can start processing on that data the bands would

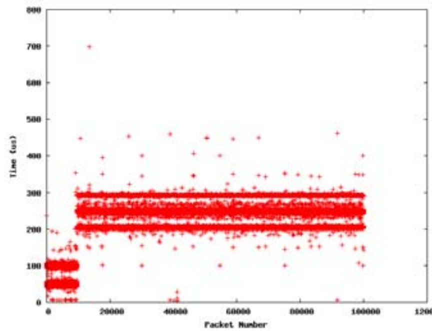

Figure 5.5: Linux

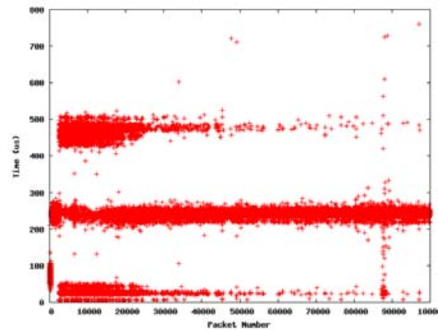

Figure 5.6: Windows XP

eventually disappear and not affect the pipeline. The behaviour of Windows XP is the most erratic in this case and thus is the worst; Windows server and Windows 7 are also quite widespread but are a lot less irregular than Windows XP. The Linux graph is the cleanest graph with almost predictable behaviour. Taking into account the 4ms and 2 ms peaks of Windows 7 and server they would still be a better choice than XP as the peaks are regular in nature and thus it can be easily programmed to take care of them.

5.2 Display Support

Every operating system offers support for a number of graphics rendering APIs. Some of these API's are hard linked to an operating system like DirectX for windows and others like OpenGL are available across many platforms. We are not interested in the 3-D graphics rendering capabilities of the system and hence the difference in the rendering capabilities of different APIs is not a major focus point. Instead we are focused on how well the API's can handle texture uploads for real time streaming.

The operating system and graphics driver play an important role in managing the data upload to the graphics card. We will evaluate the display support of the operating systems based on the efficiency by which they handle the same. To evaluate the data upload we will look at the inter frame time for the upload and the latency in uploading the image.

5.2.1 Latency Test

We want to measure the latency from the time the command to present the image on screen is issued from the program until the time the picture actually appears on the screen. We want the latency of the whole pipeline to be as low as possible for the cardiologist to have a good hand to eye co-ordination.

To measure the latency we have to measure the time difference from the issue of the present command until the time the image is actually presented on the monitor. To measure this first we developed a program, which would send continuous black images to the monitor and one white image after every 30 black images. By doing this we can easily detect this change on the monitor via a light meter. To get timing for the point where the instruction is issued to present the white image to the screen we sent a signal to the serial I/O port on the computer, which along with the light meter was then connected to an oscilloscope. The time difference between the pulse on both the light meter and the serial port gives us the latency to display of the image on the monitor. This latency also includes the latency of the monitor and the light meter in addition to the latency by the system. Since, we used the same monitor and the light meter for all the tests this latency is common to all results and can be discarded for comparison purposes.

Test with OpenGL were done for all the systems where as they were also repeated for DirectX on windows platforms.

Table 5.2: Results for the latency while using OpenGL

Image	Windows 7 ms	Windows Server ms	Linux ms	Win XP ms
1 MP @60 fps	45	45	37.8	56.5
5 MP @25 fps	61	61	54	

Table 5.3: Results for the latency while using DirectX

Image	Windows 7 ms	Server ms	Win XP ms
1 MP @60 fps	56	56	61.5
5 MP @25 fps	47	47	

For Windows XP no VSync could be achieved while uploading the 5MP texture and thus no valuable results were obtained for XP. Linux shows the best timing for the 1 MP image upload whereas Windows 7 and server are better for the 5 MP

image upload with DirectX.

5.2.2 Frame time

Frame time is the actual time difference between the uploading of two consecutive images. To understand the importance of inter-frame time we first need to look at how the monitor works. Assuming the monitor is running at 60Hz then it displays a new image every 16.66ms. This image is sent to it via the graphics card, which maintains a buffer to store the next image to be displayed on screen. Now if an image is updated in the GPU buffer anywhere in the 16.66 ms window it will be displayed on the screen at the next update. Assuming that the normal update happens in the middle of the period this gives us a window of 8 ms on both sides to upload the image. Thus if we have a jitter in our system of less than 8 ms in the worst case we can still show the image on the monitor in time and the user will not experience any jitter.

Table 5.4: Frame time while running OpenGL

Measurement	Linux	Windows XP	Windows 7	Server 2008 R2
1 MP @ 60fps				
Effective FPS	60.04	59.96	60.03	60.03
Standard deviation (ms)	0.23748 ms	1.079584 ms	0.091639 ms	0.091639 ms
Average (ms)	16.654 ms	16.677 ms	16.657 ms	16.657 ms
Min (ms)	11.896 ms	4.835 ms	16.153 ms	16.153 ms
Max (ms)	21.343 ms	70.760 ms	17.135 ms	17.135 ms
5MP @ 25 fps				
Effective FPS	25.09	24.97	25.07	25.07
Standard deviation (ms)	0.2679 ms	4.8119 ms	0.1761 ms	0.1761 ms
Average (ms)	39.851 ms	40.042 ms	39.891 ms	39.891 ms
Min (ms)	38.196 ms	27.239 ms	37.713 ms	37.713 ms
Max (ms)	41.606 ms	184.872 ms	42.812 ms	42.812 ms

In this test, we will look at the inherent jitter the system has in uploading the texture to the graphics card / monitor. Since this inherent jitter will reduce the jitter we can sustain in the pipeline it should be as close to zero as possible. Again, the test was done with both DirectX and OpenGL. The timing was taken each time the

Table 5.5: Frame time while running DirectX

Measurement	Windows XP	Windows 7	Server 2008 R2
1 MP @ 60fps			
Effective FPS	60.02477	59.97955	59.97843
Standard deviation (ms)	0.354419	0.64332	0.853066
Average (ms)	16.65979 ms	16.67235 ms	16.67266 ms
Min (ms)	14.56606 ms	8.766874 ms	9.958697 ms
Max (ms)	19.47133 ms	24.5813 ms	24.89499 ms
5MP @ 25 fps			
Effective FPS	25.04609	33.41324	33.30797
Standard deviation (ms)	1.16612	4.086599*	4.591652*
Average (ms) 3	9.92639 ms	29.92826 ms	30.02285 ms
Min (ms)	34.10809 ms	25.3147 ms	25.15104 ms
Max (ms)	72.04864 ms	38.96105 ms	43.71911 ms

function to present the image was called and the difference recorded.

From the maximum and the minimum jitter, we can see that except Windows XP with OpenGL none of the test scenarios lie outside the region of 8ms - 24ms and thus do not miss any frames for 1 MP image and same for 5 MP image also. Further, we can see that the Windows 7 and Server 2008 R2 with OpenGL have less spread than all the other scenarios we have. For the jittering, we can conclude that Windows 7 and server are the best choice with the combination OpenGL and next choice would be Linux followed by DirectX on Windows XP.

5.2.3 Monitor Handling

Another requirement from the operating system is the proper handling of multiple monitors. Not only should the application be aware of the monitor it is rendering to but also capable of handling any unexpected monitor disconnections and reconnection. This is specifically important to our application as it can have up to 8 monitors connected to it each displaying a different image. In case one of the monitors goes down any change in the position of the remaining windows might be catastrophic for the patient as the doctor might make wrong decisions not knowing the change in the image he is viewing. For this, we need the operating system to provide the full information about the monitor connections to the application and in case of any changes in the monitor configuration leave the decision to the application.

The control on the behaviour of the monitors was found to be available to the fullest on Linux platform followed by Windows XP with already acquired in-house knowledge and lastly with Windows 7 and server 2008.

5.2.4 Other Issues

It was seen that in both Windows 7 and Server 2008 there were multiple issues with VSync with OpenGL rendering. There was no VSync while using OpenGL in full screen mode and there was no VSync when more than one monitor was connected to the system. Although a bug for the same has been filed with Microsoft, it is recommended to use DirectX with Windows as it is better supported and more predictable.

5.3 Memory management

The rate of increase of computing power has left behind the memory speeds in the recent past. Increasing CPU power is relatively easier than increasing the memory bandwidth of a system. The focus in development of major systems has shifted from being CPU bound to being memory bandwidth bound [4].

Although the total memory bandwidth is limited by hardware, the available memory bandwidth in the system depends on the efficient usage of the same by the operating system. We will look at the total available memory bandwidth of the system when there is no load from any of the operating systems subcomponents and then at the memory bandwidth overhead by various subcomponents of the operating system.

5.3.1 Available memory bandwidth

The aim was to measure the total memory bandwidth available in the system in the idle mode. To measure the same a program was written which would copy, write and read data to and from a large array in the memory. The array was 100,000,000 integers long with each integer taking up 4 bytes of memory. The time taken for the copy, writes and reads was calculated and recorded. The memory bandwidth was calculated as below

Memory Bandwidth = (400,000,000 / Time taken) / (1024*1024) MBps

Single Core Test

First, a program was written which would run on a single core to calculate the total memory bandwidth. The code was compiled with all the compiler optimizations turned off. This was done to ensure that all the memory transfers did actually took place and were not skipped by compiler optimizations.

The results hence obtained were compared for different operating systems and are given in table 5.6.

Table 5.6: Memory bandwidth while using single core

Operating System	Copy MBps	Write MBps	Read MBps
Linux	2061	1226	1346
Windows XP	2105	987	1347
Windows 7	2078	986	1336
Windows server	2061	987	1336

Since we are using the same code on all the operating systems, we would expect the total memory bandwidth to be almost same for all the different OS's. From the results, it seems that in different operating systems the memory writes are handled differently. Since we are running the code on the same hardware, the difference would be in the compilers generation of the assembly code. A closer look at the code showed that one code was using move immediate instruction in the assembly code and the other code was storing the write value in a register and then writing that to the memory. The respective code segments were:

Code 1:

```
mov        DWORD PTR _j$[ebp], 0
                jmp   SHORT $LN13@main
$LN12@main:
                mov   edx, DWORD PTR _j$[ebp]
                add   edx, 1
                mov   DWORD PTR _j$[ebp], edx
$LN13@main:
                cmp   DWORD PTR _j$[ebp], 100000000
                jge   SHORT $LN11@main
                mov   eax, DWORD PTR _j$[ebp]
                mov   DWORD PTR ?A@@3PAHA[eax*4], 2
```

```
                jmp   SHORT $LN12@main
```

Code 2:

```
mov       DWORD PTR _j$[ebp], 0
                jmp   SHORT $LN13@main
$LN12@main:
                mov   edx, DWORD PTR _j$[ebp]
                add   edx, 1
                mov   DWORD PTR _j$[ebp], edx
$LN13@main:
                cmp   DWORD PTR _j$[ebp], 100000000
                jge   SHORT $LN11@main
                mov   eax, DWORD PTR _j$[ebp]
                mov   ecx, DWORD PTR _y$[ebp]
                mov   DWORD PTR ?A@@3PAHA[eax*4], ecx
                jmp   SHORT $LN12@main
```

We see that the difference in the two code segments highlighted in red is between storing the value from a register and storing the value directly. This difference is in the compilers code generation and not in the operating system. The difference in the speed of the move immediate instruction and the move register were also discussed with Intel and the same behaviour was confirmed from them.

In addition, in this case, the memory bandwidth was not the limiting factor but instead we were limited via the CPU as we were only using a single core. Therefore, the next step was to run the code on more than one core to figure out the maximum total memory bandwidth and to study how the choice of the different cores affected the outcome.

Multi-Core Test

To look at the memory bandwidth while using multiple cores we first need to check the behaviour of the system with the different selections of the cores. The hardware has dual quad-core processors with shared L2 cache between two cores. The various different configurations for the choice of two cores were both on different socket (red), on the same socket with different L2 caches (green) and lastly both on the same socket and sharing the same L2 cache (yellow).

This test was done on windows XP and was used as a study to look at the hardware division and usage of the memory lanes.

Figure 5.7: The Core Arrangement in the system

Table 5.7: Effect of selected cores on memory bandwidth

	Bandwidth MBps
Different Sockets (red)	5540
Same Socket, Different L2 cache (green)	3680
Same L2 cache (yellow)	3620

We see from table 5.7 the code running on different sockets gives us the maximum memory bandwidth. Also only a limited amount of the memory bandwidth can be accessed from a single socket. We can also note the effect of cache misses here when we try to use the sockets with the same L2 cache. The test was extended to 3 - 8 cores without any change in the total bandwidth. Thus, we can use the maximum available bandwidth by using two cores on different sockets in the system.

SSE and Final test

For the final testing, we also tried using the SSE2 memory instructions to improve the memory bandwidth. From the results, we saw that the SSE2 instructions increased the maximum bandwidth usable from one core by reducing the computational overhead but the total memory bandwidth when using multicores was the same as the real limitation was the systems memory bandwidth and not the computation power.

We see that the memory bandwidth is highly dependent on the fact that whether we are using the 32 bits OS or 64 bits OS (5.8). For 32-bit operating system, we can only address a total of 2.25 GB of physical memory as the rest is used to address the other devices and busses in the system. This limit has been set in the BIOS configuration of the system. The system uses four DDR2 ram modules, which are quad pumped (4 channel) to be able to utilize the full memory bandwidth. When using only 2.25 GB out of the total 4 GB memory the speed of the system reduces as it can be seen from the results. This is likely because the four memory banks are

Table 5.8: Total available memory bandwidth

	Bandwidth MBps
Linux (64 bit)	6700
Windows 7 (64 bit)	6400
Windows server (64 bit)	6400
Windows XP (32 bit)	5500
Windows 7 (32 bit)	5600
Linux (32 bit)	6000

not stripped equally and the total available bandwidth is not fully quad pumped.

In addition, we can see that the idle memory bandwidth available in Linux is higher than in Windows 7 / server by about 300 MBps. This difference can be credited to the operating systems overhead on the memory bandwidth.

5.3.2 Memory bandwidth overheads

To evaluate the memory bandwidth that is available for our application we must look at the memory bandwidth used up by various OS subcomponents active during the application. Looking at memory bandwidth utilization is tricky as there is no direct measure possible. The only way to measure the memory bandwidth being used, is to look at the total memory bandwidth available with and without the load running in the system. However, this is more complex than it sounds, as any attempt to measure the available memory bandwidth will have to be done by loading the system with memory copies that in turn will affect the performance of the load.

Since there is no prioritization of memory I/O, we have to make sure we do not alter the performance of the module for which we are measuring the memory bandwidth. The way to measure the idle memory bandwidth would be to put a known amount of memory load on the system and then increasing it until there is a noticeable effect on the performance of the load. This will give us the point when the memory bandwidth is full. All the tests were repeated a number of times and the results obtained were averaged. The difference between the maximum and the minimum results was less than 5% of the average result, which is relatively far, less than the differences between the different operating systems.

The loads for Disk writing, reading, displaying image and receiving network traffic were calculated.

Table 5.9: Memory bandwidth overhead

	Disk Read	Disk Write
Linux	380 MBps @ 80 MBps	200 MBps @ 69 MBps
Windows 7	550 MBps @ 78 MBps	380 MBps @ 80 MBps
Windows Server	600 MBps @ 85 MBps	400 MBps @84 MBps
Windows XP	330 MBps @ 69 MBps	260 MBps @ 76 MBps

Table 5.10: Memory bandwidth overhead

	Network Receive MBps	Display @ 60FPS Mbps
Linux	375	360
Windows 7	375	480
Windows Server	375	480
Windows XP	400	360

We see that there are three memory copies for the network traffic by the fact that there is 3 times the data flow in the memory. The same is true for uploading the textures to the graphics card for Windows XP and Linux. We see that for Windows 7 and Server there is an extra memory copy while uploading the texture, this behaviour was expected as we have an extra layer of desktop windows manager.

5.4 Device Drivers

Main factor for considering a general-purpose operating system is that it can run on a wide variety of hardware. Device driver acts as an abstraction layer between the hardware devices and the operating system or the applications using the device. A driver translates the high-level instructions issued by the program and then communicates the same to the device via a communication bus in low-level language. This enables the programmer to work without requiring having an in-depth knowledge about the underlying hardware. Also without having the layer of abstraction to the hardware, that the drivers provide the applications would have to be re-programmed for every change in the hardware.

The drivers generally run in kernel mode, which is a highly privileged environment in the operating system and thus any error caused by an ill written driver can lead to a crash. Therefore, the device drivers need to be carefully programmed and tested before they are released. The drivers should also be able to make full

utilization of the hardware by exploiting its capabilities to the fullest.

Device drivers are hardware and operating system specific. We need support from the device manufacturers for the device drivers along with bug fixes and future support. To evaluate the device driver we need to look at the official support available for the driver from the manufacturing company, the frequency of new driver updates issued for that operating system and the hardware functionalities supported by that driver.

Since Windows 7 and Windows server 2008 R2 were still not officially released at the time this analysis was done we compared the three different flavours of operating systems namely windows client, windows server and Linux. Since all the hardware companies offer similar support to the future versions of the same operating system this is a valid assumption. We will only look at the devices that need explicit driver support from the manufacturer.

Intel (Motherboard/ Chipset)

Intel support's all the three operating systems equally well with a similar frequency of updates for all the three varieties under consideration and offering full capabilities of their processors, motherboard controllers, disk controllers etc.

Graphics Card (NVIDIA)

Drivers are available for all the three varieties of operating systems for both 64-bit and 32-bit operating systems. For Windows server it is seen that the frequency of updates of the drivers is slower and thus it takes longer for the newer functionalities and bug fixes in the drivers to be available for the server editions. With Windows client and Linux, the frequency of updates is almost same with a slight delay (2-3 days) in the release time.

Ethernet Cards (Intel)

With the advanced Intel Ethernet cards, it can be seen that the support for some high-end functionalities like I/O acceleration is not available for Windows client versions. Apart from this, the support and frequency of updates is same for all three operating systems.

Table 5.11: Device Driver Support

	Ethernet Network	Infiniband	NVIDIA	Intel (NIC/Chipset)	Testing
Windows Client	Good	Poor	Excellent	Excellent	Excellent
Windows Server	Excellent	Good	Good	Excellent	Excellent
Linux	Excellent	Excellent	Excellent	Excellent	Good

Infiniband (Mellanox)

The Infiniband drivers from Mellanox are supported for the windows server editions and for the Linux, but there is no full support for windows client versions. Windows vista is not at all supported and support for windows XP has only been recently added. The support for future Linux and Windows server family of operating system is on the company's roadmap whereas nothing has been said about windows client versions. In addition, the driver offers much more functionality on the Linux version as compared to the Windows drivers like Virtual protocol interconnect.

Pre-testing of drivers

Apart from the above listed drivers, there are many other generic drivers present in the operating system kernel. Since driver errors are difficult to isolate and can affect the whole system it is important that enough testing be done prior to the shipment of the drivers. The testing methodology for Windows and Linux are quite different. On one hand Windows undergoes extensive closed testing [WHQL]; Linux undergoes open community testing. Although arguments exits that support both sides in our case we consider Windows testing more reliable than Linux.

The summary of the above results is provided in the table 5.11.

5.5 Disk I/O / File system

The file system is a method to store and organize data in the permanent memory. The filesystem module of an operating system handles the file input and output operations. It gives access to the stored information on the system and provides the necessary interface to store and retrieve data. On top of the basic archiving and

retrieval of data, filesystem's offer many more functionalities like file hierarchy, journaling, hard links, symbolic links, execution in place etc.

For disk I/O tests, the tool IOzone was used which is available for both Windows and Linux platform. IOzone filesystem benchmark provides benchmarks for variety of file operations and is available under many platforms making it a good choice for file I/O benchmarks. For our evaluation, we are mainly interested in measuring the file read and write bandwidth along with the CPU overhead for single and multiple threads of access. We will also evaluate the system performance on a software raid zero, implemented to increase the overall speed of the system.

The optimality of the filesystem depends on both the operating system and the filesystem chosen. There are many choices available for choosing a filesystem, each with their own advantages and disadvantages. We will only investigate the major filesystem for each operating system. Furthermore, we will try to isolate the filesystem choice from the results by running tests with the same filesystem over different operating systems.

5.5.1 Read / Write Bandwidth test

While measuring the read and write bandwidth of a file system we have to ensure that there is no other disk I/O operation running on the disk in the background. To ensure this we require that there are no swap files or system files on the disk to be checked. Secondly, the caches and the buffers also have to be accounted for, while writing small files the operating system confirms the write when the data is still in the caches or the memory. It was seen that for files of smaller size the data was still in cache somewhere while the system reported that the data was written. The caching / buffering of the data can happen either in the main memory or in the hard disk buffers. Even after issuing a call to close and explicitly flush the data, the results obtained showed that the data was still not written to the hard disk. To avoid the same all the tests were thus done with large file sizes (6GB each). This ensured that the data was written to the hard disks and the effect of the data still in the disk buffers was minimal.

The following command was used to get the performance benchmarks from the IOzone tool.

```
iozone -c -t 1 -r 4M -s 6G -i 0 -i 1
```

```
-c ensures that we include the fsync and close timing
 in the measurements.
-t specifies the number of consecutive threads to run
 on the system we ran the test with one two and three
 threads
-r is given to specify the chunk size 4MB in our case.
-s gives the size of the file (6 GB)
-i specifies the tests to run we chose read and write
bandwidth tests.
```

It was found that the reruns of the same test would produce different throughput figures with variations up to 15% under the same environmental conditions. This was because we were working on the assumption that the disk speed is constant irrespective of the location where we are writing on the disk. In reality, the disk speeds vary by 35% depending on if you are writing to the start of the disk or towards the end 5.8. It was impractical to direct the operating system to write the data on a specific physical location on the disk for each operating system. Therefore, a smaller partition sufficient for the test was created at the start of the disk.

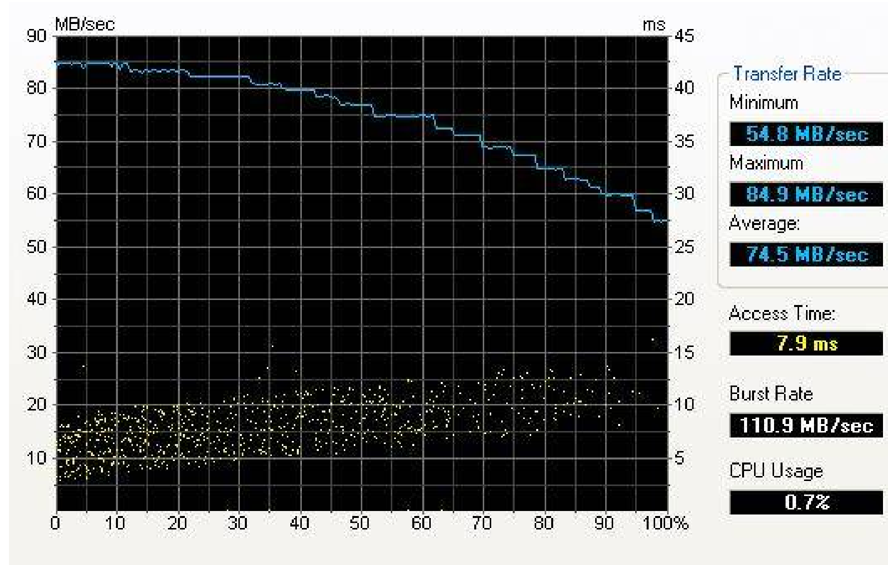

Figure 5.8: The disk access speed with position

This ensured that the physical speed of the disk remained consistent between tests. The tests were repeated 3 times for each operating system and an average

value of the results was taken. The difference between the consecutive tests run was less than half a percent. The tests were first done with each operating systems native filesystem for single double and triple threads of access. That is NTFS was used for Windows XP, 7 and server 2008 R2, whereas ext3 was used for Linux.

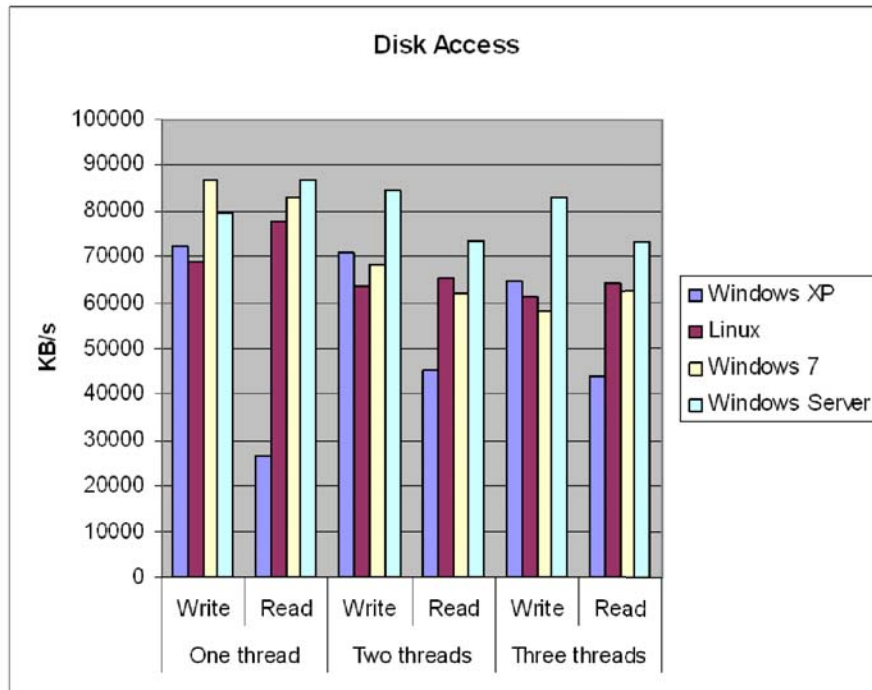

Figure 5.9: The disk read write speeds

We see from Graph 5.9 that Windows server 2008 R2 gives the best performance in all the cases. Linux and Windows 7 are close behind. We see a gradual decrease in performance of Windows 7 as the number of threads increase whereas the same is somewhat consistent for Linux. We also see here that the write bandwidth on Linux is lower than that of the other operating systems; this can be accredited to the fact that ext3 uses journaling and maintains a journaling log for all the file write operations. While the reason for the poor read performance of Windows XP could not be pinpointed, we did learn from Microsoft that the Windows 7 and server 2008 R2 did have a new disk I/O API.

Since the results above are both filesystem and operating system dependent, we tried to separate the filesystem from the disk access in the operating system. This was attained by running different filesystem's on each operating system. We used

the drivers available for ext2 on Windows XP and the ntfs-3g drivers for accessing NTFS on Linux.

Table 5.12: Total speed while using one thread

Operating Systems	NTFS Write KBps	NTFS Read KBps	EXT3 Write KBps	EXT3 Read KBps	FAT Write KBps	FAT Read KBps
Linux	44492	83991	68942	80608	81813	92580
Windows XP	72241	26531	76238	52696	74451	85313
Windows 7	86710	83130	-	-	94007	81095
Windows Server	79384	86638	-	-	120437	87173

Table 5.13: Total speed while using two threads

Operating Systems	NTFS Write KBps	NTFS Read KBps	EXT3 Write KBps	EXT3 Read KBps	FAT Write KBps	FAT Read KBps
Linux	35407	33460	63548	65578	35251	45746
Windows XP	71005	45559	38964	18927	70588	28606
Windows 7	68069	61970	-	-	61103	60727
Windows Server	84462	73343	-	-	85977	73156

Table 5.14: Total speed while using three threads

Operating Systems	NTFS Write KBps	NTFS Read KBps	EXT3 Write KBps	EXT3 Read KBps	FAT Write KBps	FAT Read KBps
Linux	35517	34236	61357	64220	22958	44073
Windows XP	64911	44076	33695	16884	62907	38608
Windows 7	58510	62621	-	-	50233	61930
Windows Server	82989	73150	-	-	83899	72466

We see that the results obtained in tables 5.12, 5.13 and 5.14 are not in co-relation with the original results. This can be expected as now the performance also depends on the implementation of the file system drivers that were installed to access the non-native partitions. Another try was made by using FAT32 file system as we imagined that the relatively simpler implementation of the FAT32 and its long

age could have resulted in having mature and similar drivers for all the operating systems but again the results proved otherwise. Henceforth no conclusion can be drawn from the results by using the non-native filesystem.

Since the file system is also an inherent part of the operating system the results for disk access API in the operating system and the filesystem can be looked upon as a consolidated result.

Software RAID Bandwidth test

Two disks were used to create a software raid 0 to increase the disk access speeds. The tests were done with each operating systems native filesystem for single double and triple threads of access. That is NTFS was used for Windows XP, 7 and server 2008 R2, whereas ext3 was used for Linux as used for the first test.

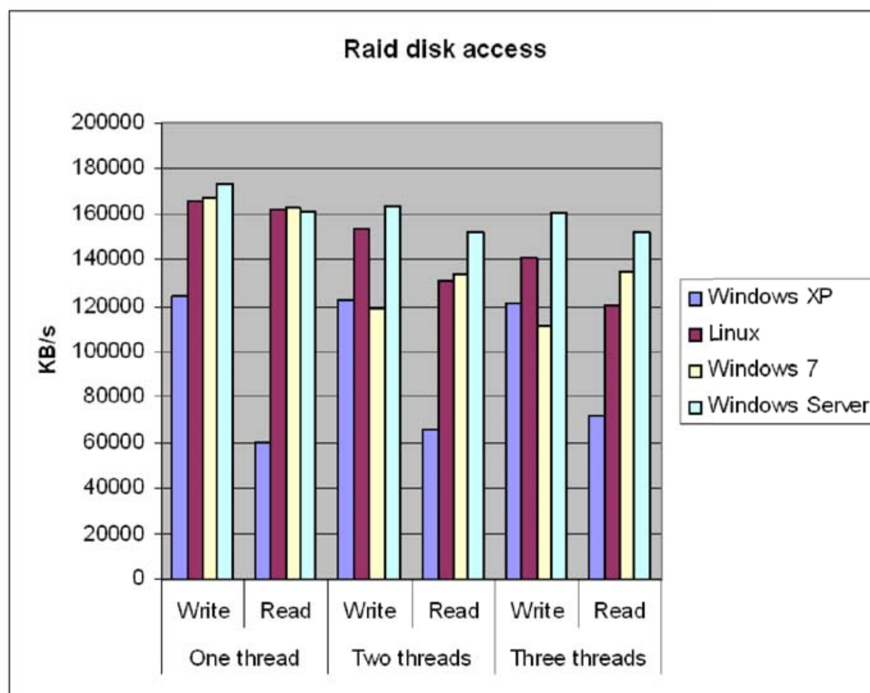

Figure 5.10: The disk access speeds while using RAID 0

We see from Graph 5.10 that again Windows Server 2008 R2 has the best performance statistics. This time the write speed for Windows 7 suffers a great performance setback with more than one thread accessing the filesystem. The major change in the performance of RAID over the normal disk performance from the

Table 5.15: CPU overhead for reading/writing to disk

Operating Systems	CPU usage (Write%)	CPU usage (Read%)
Linux	17	7.5
Windows XP	2	2
Windows 7	2	1
Windows Server	1.5	1

viewpoint of the operating system would be handling of multiple threads. For a single thread, the operating system would perform simultaneous writes on both the drives but when writing multiple threads proper dividing of work can significantly reduce the seek time delay from switching from one file to another. From this graph we see that while the performance of Windows 7 and Linux is same without RAID, Linux has an edge over Windows 7 in write speeds while using raid.

5.5.2 CPU overhead

The computational overhead of the disk I/O was also observed and since the same utility (IOzone) was used for all the operating systems its overhead was neglected.

It was noticed that the CPU utilization was exceptionally high on Linux where as it was nominal on all Windows variations. The extra overhead in Linux can be accounted due to two things; firstly, ext3 supports full journaling meaning that it writes all the actions it is going to take and then after successful completion marks them as successful and second which is more compute intensive is the use of inodes. Linux uses inode structure to store files and thus has to load the inode table and looks into the entries to find a position to store or to find the location of the stored file. On the other hand, this ensures that the disk system does not fragment which is not the case with NTFS.

5.6 Scheduler

Scheduler is responsible for maintaining the run time queues for the different cores and deciding which process gets the time slice next. Scheduling also ensures that the real time tasks can meet their deadline while the lower priority tasks wait for their turn. Theoretically, the scheduler for windows 7, windows server and Linux 2.6.28 are O(1) in complexity and that of windows XP is O(logn) in complexity.

Due to the closed source of windows, it was not possible to measure the scheduling latency of the system. One of the important factors for a system is the context switch time. For a system to be able to able to respond in real time the system must have a small and constant context switch time. We will look at the context switch times for each operating system.

5.6.1 Context switch time

Every time the operating system swaps out a current process to make room for a new one, or an interrupt occurs or if it needs to go from user mode to kernel mode or vice versa a context switch takes place. There are about tens of thousands of context switches occurring in an operating system per second. Therefore, even a small difference in the time between two operating systems has a considerable effect on the overall performance.

A context switch involves saving and loading of the following values in hardware [13].

- Instruction pointer
- User and kernel stack pointers
- A pointer to the address space for the thread
- The registers

Test 1

To measure an context switch first a program was written which would start two processes one of real time priority and other of lower priority in suspended mode and then resume the lower priority process first and then the higher priority process and measure the time taken by the higher priority process to start executing. Since the overhead of the process is much higher than the actual time taken by the switching no meaningful results were obtained.

Test 2

The next test was done with having two processes of equal priority running in an infinite loop and yielding their time slice while taking timing measurements. The goal was to find the smallest time difference between the two processes. Again, the overhead and the granularity of the timing function were not sufficient to measure the time taken by the context switch.

Table 5.16: Context switch times

	Time (us)
Linux (64 bit)	1.35
Windows 7 (64 bit)	1.45
Windows Server (64 bit)	1.45
Windows XP (64 bit)	0.84
Windows 7 (32 bit)	0.89

Final Test

The final test was done by using hardware counters to measure the number of context switches occurring on a particular core per second and having a code running an infinite loop and yielding its time cycle forcing context switches to occur. This value was then used to calculate the time it takes for a context switch. The hardware counter was read with the help of Vtune a performance analysis tool from Intel.

The difference between the context switch times for the operating systems in the fact that XP is 32 bit whereas the other operating systems are 64 bit based. For a 64-bit operating system, the number and the size of registers increase which accounts for the additional time it takes a context switch to save and load the extra information. This was also confirmed by running the test for 32-bit version of Windows 7 and the time was found to be 0.89us, which is comparable to 32-bit Windows XP. Although the time difference in the two systems is just 0.6 us if we look at the number of context switches occurring in a system per second which is around the order of 10,000 We will see that this difference translates to the order of 0.6 % CPU overhead. This is summarised in the table 5.16

5.7 Reliability

The IEEE definition of reliability is ”the ability of a system or component to perform its required functions under stated conditions for a specified period of time”. Each system requires a certain degree of reliability from the system. The extent of reliability expected depends on the application of the system.

A system can have both hardware and software faults. The operating system should be able to safeguard itself against any attack or faults that might occur. The operating system should enable handling of hardware errors and at the same time being free from any software errors. Since it is rarely possible to have a complex

system like an operating system bug free the system should be able to handle errors in a respectable fashion or in other words it should have a graceful degradation. The system should be able to adequately locate the point of error and have support for proper logging and debugging of the issue.

The measure of reliability is an abstract quantity and thus not easy to measure [11]. A system can run on for years before a flaw is discovered. To compare the reliability of an operating system we looked at the published statistics of the number of known vulnerabilities of the various operating systems. Another important factor to be considered is the maximum uptime of the system, or the period the system can run without requiring a reboot. Bugs per number of lines of code is also a commonly used method for evaluating the reliability of a piece of software but we will refrain from using that as the windows is closed source and only speculative numbers are available for the lines of code and for number of expected bugs.

The published uptimes of the various operating systems vary a lot based upon the underlying hardware and the intended use of the system. Since it is not a fair comparison comparing uptimes of operating systems in different environments and not feasible to measure the uptime by our own in the period of the thesis we will not use the operating system uptime as a measure for reliability.

Therefore, we had to limit ourselves on the number of known bugs both reported and outstanding for an operating system. We used the statistical data on different operating systems from Sceunia a security advisory company. The statistics are provided for the total number of security bugs reported and the number of outstanding issues. These are further classified with their severity ranging from extremely critically to not critical.

The table 5.17 gives the total number of known advisories for a number of operating systems. It is not sufficient to just look at the total number of known advisories as it only comments on the total number of reported bugs and is dependent on the life of the operating system. A better measurement is the number of un-patched advisories and the number of reported extremely critical cases. We see that for the Linux distributions these two numbers are zero, which is quite good. In addition, the highly critical cases run in 30%'s for the windows systems whereas they are lower for the Linux systems. We can also easily see that the number of total bugs is directly proportional to the lifetime of the project. Thus, we look at the percentage of the un-patched and the extremely critical advisories.

So we can conclude from our observations that the Linux is the most reliable system, followed by Windows server followed by Windows client versions.

Table 5.17: Secunia statistics on vulnerabilities

Operating System	Unpatched Advisories	Total Advisories	Extremely Critical	Highly Critical
Windows XP pro.	14%	231	4%	34%
Windows Vista	10%	51	2%	33%
Windows server 2003	8%	180	4%	37%
Windows server 2008	8%	26	0%	31%
Fedora Core 9	0%	194	0%	16%
Ubuntu 8.04	0%	84	0%	26%
Ubuntu 8.10	0%	38	0%	24%

Chapter 6

Consolidation Methodology

We tested the various individual functionalities in an operating system. Now we need to develop a model that will give us a score on the overall system performance from the individual results. This model can then be used to calculate the performance of our application running on the various operating systems.

6.1 The model

We proposed in our hypothesis that the application performance can be evaluated as a function of various resources used by the system while running the application and the total hardware resources available to it. We will now develop a model which will give us the total hardware resources used ny the system.

The hardware resources available in the system are shown in the figure 6.1.

The main memory bandwidth is limited at the north bridge and finally the CPU computation power is limited by the processors. We use the percentage CPU utilization used to look at the CPU power to ensure that the stalls are also included and not just the number of instructions being executed on the CPU. Next, we have the memory bandwidth limitation at the Northbridge we looked at the total available memory bandwidth in previous chapters for each OS. Similarly, we also have computed the various speed limitations for the Disk, Network card and the Graphics card.

The resources used by the system can be classified into two main categories; one used by the operating system to facilitate the application including the application interaction to the operating system and the second as the algorithmic load of the application itself. Since, the application logic is independent of the operating system the overhead caused by this would solely depend on the way of programming

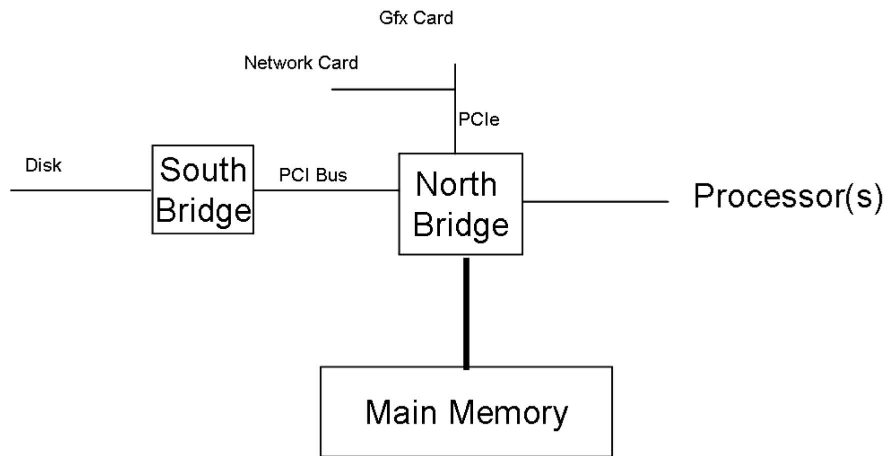

Figure 6.1: The system Architecture

on different operating systems and the compiler used leaving the application logic overhead constant for each operating ssytem. As we saw that the core performance of the system remains unchanged with the choice of operating system it would mean that if we ran the same assembly code on each operating system we would get the same resource usage. The only difference would be in the performance of the OS functions called and the other OS services running in the background. Thus the variable part would be the application induced operating system overhead. This means that to compare the operating systems we can solely look at the OS overhead part and not worry about the application logic load.

We know from our study of the operating system structure in Chapter 2.1 that the operating systems tested have a modular monolithic structure. This implies that all the modules of the operating system function as independent applications. These modules interact only via the kernel and are independent of each other. Now when running two different modules on the operating system the OS will run them as two tasks running at the same time and switch between those to enable multi tasking. Now as long as the two modules or tasks do not contend for a common resource they will run fluently and not affect one another. In case of contention there will be idle cycles in the CPU due to the wait cycles for the resource which will increase the CPI of the system thus inducing extra load. In an application with multiple threads of processing the operating system will always try to keep the hardware resources busy by scheduling the threads which are not waiting on a resource. The overhead caused by waiting on resources will be negligible in cases where the limiting resource usage is not high enough as the scheduler can

schedule tasks accordingly. But when the resource usage will approach its hardware limit the contentions will increase drastically inducing additional operating system overhead; mainly in terms of scheduling latency, context switches, and idle wait cycles. Also when approaching the hardware resource limit due to the many contentions the application behaviour will become unpredictable and un-suitable for use in real time applications. Thus when using more than one component block at the same time and before reaching hardware limit for any resource the overall hardware usage of the total system can be computed from the individual hardware usage of the different modules added to the hardware resource usage by the kernel itself.

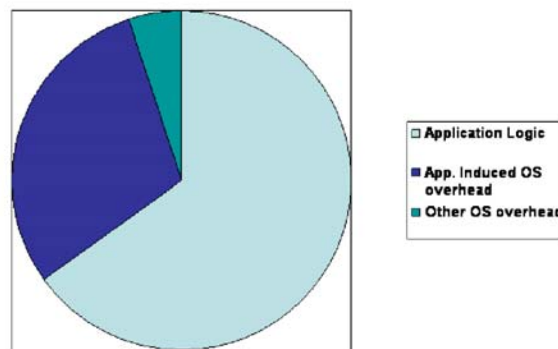

Figure 6.2: Resource load on the system

Thus we can now compute the load on the various hardware resources of the system given the load by the different OS modules and the application load. We have already studied the load generated by various OS components in the previous chapter and can thus predict the load given the input load on the functionality. To analyze the system we looked at the smallest basic unit of execution for our application which has a linear load characteristic. The C/V image processing application works on processing image frames. The basic unit of our application would be a single image frame. Since the load on the system is dependent on the throughput of the system, we will take the unit to be one frame per second.

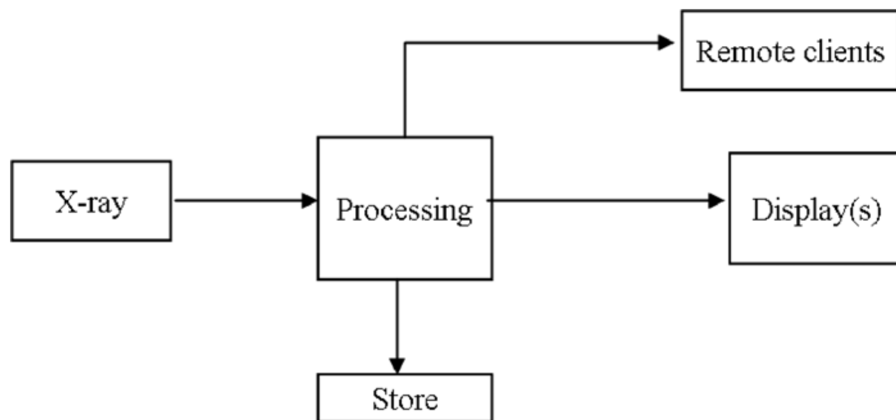

Figure 6.3: The processing chain

6.2 Performance Evaluation

In the pipeline, we can see that the overheads are from the following components per frame along with the kernel overhead

1. 1 Network receive
2. 1 Disk write
3. 2 Display
4. 1 Network send
5. 1 Application Processing

We will calculate the hardware overheads from the four OS components for the various hardware limitations namely memory bandwidth, CPU and the I/O and as the application processing has been seen to be constant for all the operating systems we will measure it at the end.

Since in the processing pipeline we have the receiving of the data packets in bursts for each frame 6.4 and similarly we have the store and display occurring for each image at the maximum possible rate we have the loads in bursts as shown in the graph for each module. These loads from each module will add up to contribute to the total overhead. However, as the total load starts approaching the maximum total available resources the performance will go down as the modules will have to wait on each other and this will result in stalls. This will only occur for cases where the load is approaching maximum available resource. For a real time application,

this would also mean that the performance is not reliable anymore and the predictability will go down. This behaviour is common to all the operating systems and thus for relative comparison it can be ignored.

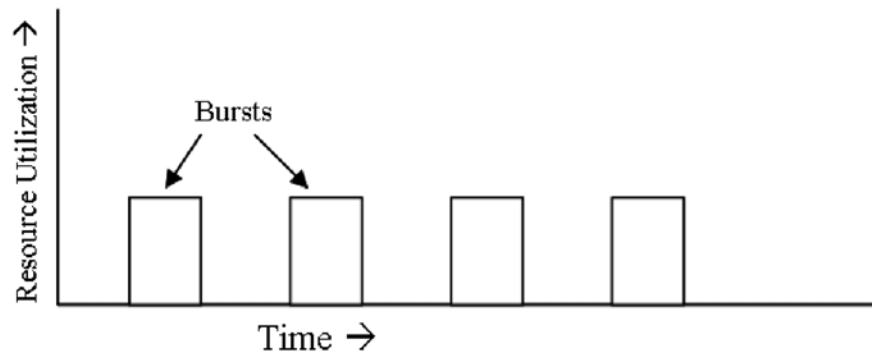

Figure 6.4: The load on the system due to each functionality occurs in bursts

Now we will calculate the various overheads as percentage of total available resource with taking in consideration only the linear behaviour of the application. Detailed calculations are not shown for the sake of clarity. The results can be obtained from the earlier results.

6.3 Results

6.3.1 Memory Bandwidth

Each image frame is of size 2MB thus we can calculate the available memory bandwidth by adding the memory overhead for each component per 2 MB frame.

Memory Bandwidth utilization per frame =
1 * Memory utilization from network receive
+ 1 * Memory utilization from disk write
+ 2 * Memory utilization from display
+ 1 * Memory utilization from network send

6.3.2 CPU usage

CPU utilization per frame =
1 * CPU utilization from network receive
+ 1 * CPU utilization from disk write

Table 6.1: Memory bandwidth utilization per frame

	Bandwidth Utilization MBps	Bandwidth Utilization %
Linux	29.8	0.45
Windows 7	37.5	0.58
Windows Server	37.5	0.58
Windows XP	31.6	0.57

+ 1 * CPU utilization from network send

+ 2 * * CPU utilization from display

+ Kernel overhead in terms of context switches

Table 6.2: CPU utilization per frame

	CPU Utilization %
Linux	0.367
Windows 7	0.273
Windows Server	0.217
Windows XP	0.284

6.3.3 I/O

Disk I/O. network I/O and Display I/O is same for all the operating systems and is equal to the I/O utilization per frame, which is 2 MB per device as there is a different graphics card for each stream and similarly different network card for sending and receiving.

Table 6.3: I/O utilization per frame

	Disk I/O %	Network I/O %	Display I/O %
Linux	1.23	1.6	0.03
Windows 7	1.21	1.6	0.03
Windows Server	1.15	1.6	0.03
Windows XP	1.60	1.6	0.03

6.3.4 Application Overhead

Here we list down the actual computation overhead of the application in terms of CPU load and memory load per frame

Table 6.4: Application logic overhead per frame

	CPU load %	Memory Bandwidth MBps	Memory B/W %
Linux	1.6	125	1.865
Windows 7	1.6	125	1.953
Windows Server	1.6	125	1.953
Windows XP	1.6	125	2.272

Thus, we can have the total overheads for the whole system by adding the application overheads to the application induced operating system overhead

Table 6.5: Total resource utilization per frame

	CPU load %	Memory B/W MBps	Memory B/W %	Disk I/O %	Network I/O %	Display I/O %
Linux	1.967	154.8	2.310	1.23	1.6	0.03
Windows 7	1.873	162.5	2.539	1.21	1.6	0.03
Windows Server	1.817	162.5	2.539	1.15	1.6	0.03
Windows XP	1.884	156.6	2.847	1.60	1.6	0.03

We can see that the application is memory bandwidth limited for all the operating systems and we can thus calculate the maximum frames per second possible on each operating system by calculating for 100% memory bandwidth load.

Table 6.6: Maximum frame speed achievable

	Max FPS	CPU load (theoretical) %
Linux	43.29	85.15
Windows 7	39.38 7	3.75
Windows Server	39.38	71.55
Windows XP	35.12	66.16

To compare the operating systems we will also look at the resource utilizations

at 35 FPS in the Table 6.7

Table 6.7: Resource utilization for frame speed of 35

	Memory Bandwidth %	CPU load %	Disk I/O %	Network I/O %	Display I/O %
Linux	80.85	68.84	43.05	56.0	1.05
Windows 7	88.87	65.55	42.35	56.0	1.05
Windows Server	88.87	63.56	40.25	56.0	1.05
Windows XP	99.65	65.94	56.0	56.0	1.05

We see that there is a performance increase of 11.5% in terms of maximum achievable FPS from moving from windows XP to windows 7 or server and an increase of 22.5 % when moving to Linux. From this we can conclude that the performance gain from the choice of operating system is quite significant. Part of this gain comes from the fact that we have more memory bandwidth available on the 64-bit platform. If we compare Windows 7 and Linux we see that Linux has a performance gain of 10% which is purely due to the difference in the operating systems.

We see that the operating system overhead or the load due to the functionalities provided by the OS is approximately 20-28 % of that of the total load. Thus any performance gain in the operating system leads to direct gain in the performance of the whole application.

From the theoretical CPU load we can see that the application is still not CPU limited at 100% memory bandwidth for any of the operating systems. In practice the CPU load will also reach 100% as the memory load reached 100% due to the fact that the cap on the memory bandwidth will lead to CPU cycles stall and thus increasing the cycles per instruction(CPI) of the system. The CPU cycles can be used to implement more computations in the algorithms without increasing any additional load on the system as then the stalls would be used for computing. We can also expect unpredictable system behaviour at memory load approaching 100% as the different modules fighting for the bandwidth will have to wait on each other and will introduce latencies in the chain.

6.4 Validation

To validate the results obtained we compared them with the CPU usage of the current system running Windows XP at various frame speeds. The results were

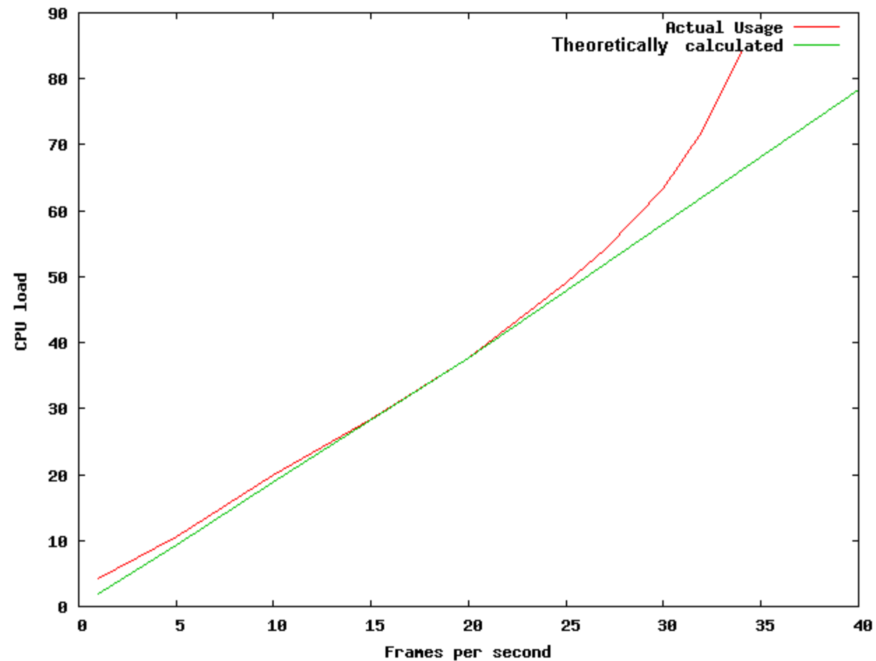

Figure 6.5: The measured CPU load vs the fps for Windows XP system

plotted against the predicted results from the model above. From the graph 6.5 we see that the predicted results follow quite closely to the actual results with a error margin of $<3\%$ obtained till about a load of 28 fps and then we enter the non linear behaviour of the system. The higher CPU usage for lower frame rates is due to the various overheads in the system which become negligible as the frame rate increases. In our model we assume that there is no resource usage for processing 0 fps but that is not the case in practice, thus we see that while our predicted model crosses the origin while the actual results do not.

Also the CPU usage can be seen to increase exponentially as the load on the system crosses a certain threshold. This is the point at which the memory contentions in the system become considerable and thus the CPI of the system will go up. Also beyond this point the system will cease to be predictable and is thus undesirable for our system.

We see that the model fits perfectly for Windows XP, and from this we can infer that it will also work for the other tested operating systems as they have the same structure. While building the model we had made two assumptions; the first one was that the resource utilization of independent components of the operating

system can be added to each other for an operating system which has monolithic structure and secondly that the load on the system will be linear with reference to frames processed per second. By validating our model for windows XP we see that both our assumptions hold true for windows XP. Since the first assumption is valid for all monolithic OS's this will hold true for all if it holds true for one. Secondly, the assumption that the load is linear with the frames processed per second mainly concerns our application and is true irrespective of the underlying operating system.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude our thesis work. The first part provides the conclusions drawn from our research. Next, we look at the various directions to extend our research in the future.

7.1 Conclusions

With our research we developed a methodology to evaluate an operating system to predict the performance of an embedded application running on it to help the developer make an informed choice for choosing the right operating system for that application. Currently a developer needs to either make the decision based on his prior experiences and knowledge of the operating systems or by coding the application for different operating systems. This leads to either making an uninformed choice or by having to spend too much time in development. Our methodology makes it possible to predict the applications performance behaviour without having to code the application for each operating system.

Knowing the predicted behaviour of an application can really help the developers in making the right choice for the operating system. This also helps in knowing in advance what the choice of a specific operating system would imply in terms of effort required in different areas. For example, if a choice of operating system predicts that the application will be I/O bound then the developer knows before hand that choosing that operating system would mean that the application has to be coded in such a way to save on I/O where as the other resources can be expended more freely and thus optimize the application better for the operating system. Similarly having studied the operating system for the application can help point out the expected issues to be faced for ensuring predictability or ensuring real time

guarantees.

We formulated the approach to evaluate and score an operating system to predict the behaviour of an embedded application running on top of it. We started with looking at the method to predict the application behaviour by micro benchmarking the operating systems various functionalities. The results thus obtained were merged together to give a final score for the operating system and validated by comparing them to the actual results obtained from running the application on Windows XP.

First, to decide the importance of the various factors in an operating system the applications requirements were studied from the viewpoint of the operating system. The IP pipeline is a high performance and a real time application. Among the various requirements memory bandwidth, networking, display support were most important functionalities.

The next step was to test the various functionalities in the operating system in isolation from other aspects. Platform independent low-level tests were developed for the same and used to test the operating system. Development of the tests required knowledge of the internal workings of the operating system and that was not always possible. For testing of functionalities which are close knit with hardware we had to work with a lot of variables. Apart from not knowing what is happening in the operating system there is also a big abstraction on the hardware. The workings of the OS/ hardware had to be reverse engineered by looking at the test results to be able to understand and look at the operating system's influence on the hardware.

The tests looked at the system behaviour for a particular functionality. It was found that most of the hardware limitations remain unaffected by the choice of operating system. The major difference in the operating systems was not in the core kernel but in the module layer built around the kernel like, file system, network stack etc. We have summarized the performance difference of the various functionalities in the table 7.1 with 5 being the best and 1 being the worst score.

To consolidate the results a linear model was developed based on the basic computation unit for the application i.e. one image frame. Since all the computation was based around the number of image frames arriving per second, the load on the system increased linearly with the increase in the frames per second. The performance of the operating systems was evaluated by looking at the various hardware resources used by the application and by finding the limiting factor.

The resource utilization for this basic unit of execution for each functionality used by the application was thus added and the total overhead by the operating systems components was calculated. This along with the resources used by the

Table 7.1: Short summary of individual results

	Linux	Windows 7	Server	Windows XP
Networking	5	3	4	1
Display	4	3	3	5
Memory Management	5	4	4	2
Predictability	5	3	3	3
Disk I/O	2	4	5	3
Scheduler	3	3	3	5
Reliability	5	4	5	4
Drivers	5	3	4	3
Overall	34	27	31	26

application logic gave the maximum throughput possible on each operating system for the application.

From our results we saw that Linux was the best OS with performance exceeding by 22.5 % from Windows XP and also the performance of windows 7 and server was better than XP by 11.5% for the tested application.

7.2 Future Work

To conclude this thesis this section provides a wide array of indications for future research. The future work is divided into two sections one focusing on the OS evaluation model and the other focusing the OS evaluation for the IP pipeline provided in the appendix C.

7.2.1 OS evaluation model

Optimization of the operating systems

The operating systems can be optimized to function better in one aspect at the cost of others. Thus optimizing the operating systems to favour the overall application before testing would yield in better results. The optimization of the operating system for a particular application would require a small study of its own and would aid in the decision process. For example in our application, we see that the system in memory bandwidth limited and hence we can optimize the operating system to save on memory bandwidth at the cost of extra computation or I/O for example.

To increase the performance of the system one could look into saving on the

memory copies happening in the IP stack. One of the options would be to use a zero copy IP stack which would give us a saving of 2 memory copies per network stream (assuming it still needs one memory copy) giving us an performance boost of 4.5

Developing the whole set of micro-benchmarks

The development of the micro-benchmarks in the thesis was focused on the requirements of the IP pipeline. These benchmarks were sufficient for the IP application but we might require a different set of benchmarks for different application. A proper set of platform independent benchmarks will aid in faster evaluation of OS for future applications. Tests could be developed for functionalities like inter process communication, interrupt handler, resource allocation threading support etc.

Extension to compute the hardware requirements

The development model for most applications starts with expected output requirements and the aspects like OS, hardware are decided along the development process. This study can be further extended to evaluate the type and amount of hardware needed along with evaluating the best operating system for the application. The research would then also involve looking at the different hardware architectures based on the identified requirements. For example, depending upon if the system is memory limited or if it is I/O limited, we can make the choices for hardware accordingly.

Bibliography

[1] Chronic diseases, the power to prevent, the call to contro. Technical report, National Center for Chronic Disease Prevention and Health Promotion, 2009.

[2] Saurabh Bagchi. Software fault tolerance on mars: Mars pathfinder 97 story, 1998.

[3] Gunter Bolch and Stefan Greiner. Performance evaluation of operating systems using approximate analytical methods. In *Conference Proceedings of the ESS94European Simulation Symposium 94*, 1995.

[4] Doug Burger and James R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, 1996.

[5] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 120–133, New York, NY, USA, 1993. ACM.

[6] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. Tcp performance re-visited. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and design of embedded systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[8] Dylan Griffiths and Dwight Makaroff. Hybrid vs. monolithic os kernels: a benchmark comparison. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 30, New York, NY, USA, 2006. ACM.

[9] Wolfgang A. Halang. Real-time systems education: What is really essential? *Real-Time Systems Education Workshop, IEEE*, 0:156, 1998.

[10] Randall S. Janka and Linda M. Wills. Early system-level design exploration of large dsp systems targeted for real-time embedded cots multiprocessors. In *COTS Multiprocessors, International Conf. Signal Processing Applications and Technology and DSP World Workshops (DSP WorldICSPAT)*, 1999.

[11] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *In Symposium on Reliable Distributed Systems*, pages 72–79, 1997.

[12] David L. Levine, Sergio Flores-Gaitan, and Douglas C. Schmidt. An empirical evaluation of os support for real-time corba object request brokers, 1999.

[13] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[14] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 221–231, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. In *PAM*, pages 247–256, 2004.

[16] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[17] J. A. Stankovic and K. Ramamritham. The spring kernel: a new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, 1989.

[18] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1993.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[20] J. Bradley Chen David Mazires Kee Chan Antonio Dias Yasohiro Endo, Michael D. Smith and Margo Seltzer. The impact of operating system structure on personal computer performance, 1995.