Vrije Universiteit Amsterdam

Parallel and Distributed Computer Systems

Efficient Use of Heterogeneous Multicore Architectures in Reliable Multiserver Systems

MASTER THESIS

Valentin Gabriel Priescu

student number: 2206421

Supervisors:
Herbert Bos
Tomas Hruby

Second reader:
Dirk Vogt

August 2012

Abstract

While designing operating systems, we often have to make a compromise between performance and reliability. This is not a problem anymore, as we can take advantage of today's powerful multicore platforms to improve performance in reliable systems. However, this comes with the cost of wasting too many hardware resources and energy. This thesis discusses techniques to make fast and reliable operating systems better utilize their resources. Therefore we perform a case study that extends a previous implementation of a very reliable and fast OS subsystem (the network stack), which wastes too many cores. We analyze if OS components really need powerful cores and if we can replace them with smaller low power cores. We simulate these small cores using commodity hardware and frequency scaling. This work also presents performance evaluation for different frequencies, discussion of different hardware capabilities that can be used to improve system efficiency, and compares with other related work.

Contents

Abstract i

1 Introduction 1

2 Background 5
2.1 What are Fast-Path Channels? 5
2.2 High Throughput Networking Stack 7

3 Implementation 9
3.1 ACPI driver 10
3.2 P-states and frequency scaling driver 15
3.3 Core-expansion/consolidation for OS Components 16

4 Evaluation 18
4.1 Experimental Setup 18
4.2 Low Power Cores on AMD Opteron 19
4.3 TCP/IP Stack Consolidation on AMD Opteron 22
4.4 Low Power Cores on Intel Xeon 24
4.5 TCP/IP Stack Consolidation on Intel Xeon 26
4.6 Discussion 26

5 Related Work 29

6 Conclusions 31
6.1 Summary 31
6.2 Directions for Future Work 31

Chapter 1

Introduction

Until recently, the research question *How can we design a fast and reliable operating system?* did not have a satisfactory answer. The answer revolved around two design types, but none of them were able to deliver both performance and reliability. The two types include monolithic operating systems and multiservers.

A multiserver is an operating system composed of multiple servers (and drivers) that run on top of a thin layer of privileged code called microkernel. Each server is responsible to implement a specific service which is used by applications or other servers. All servers live, like normal processes in userland, separated by address spaces. Due to this design, a multiserver is said to be a more reliable operating system. The reason is its ability of isolating components, restarting and replacing them on the fly, without interrupting the entire system. The microkernel exports an Inter-Process Communication mechanism based on message passing which is used by the servers to talk to each other. This is the element that deemed multiservers slow, because of the large number of IPCs and their high overhead.

In contrast to multiservers, a monolithic operating system (e.g. Linux, Window), as the name suggests, bundles all operating system components into a single binary that runs in kernel mode. The inclusion of all services in kernel space has some drawbacks: the size of the kernel code is huge (it exceedes 15 million in Linux 3.0), which makes it hard to maintain and to debugg, does not have isolation between components (a faulty driver can crash the entire system). The main advantage of this design is the performance. All OS components can communicate very fast through procedure calls.

Previous research projects tried to answer the question by improving the performance on multiservers. For example, the L4 microkernel [1] uses hardware registers and better scheduling decisions to increase the performance of the message passing communication system. However, this is still far from the performance of a monolithic operating system. Others tried to make monolithic operating systems reliable. Nooks [18] isolates each Linux driver into a lightweight wrapper to prevent errors from propagating to other OS components, but this applies only for drivers. It does not isolate core components, like the network stack.

Regardless of their type, operating systems share the CPU between applications and OS components. Every time a context switch(change execution between two processes) is made, caches, TLBs and branch predictors are flushed and a new context (for the new process) is loaded. These cancel most of the complex features that modern processors have to offer for improving performance [17]. Recent research use multicore processors and focus on dedicating some of the CPU cores to OS components to avoid all these context switches and to keep caches, TLBs and branch predictors warm. IsoStack [16] uses dedicated cores for an AIX[1] network stack, and leaves the rest of the cores for applications. T. Hruby et. al. [7] introduce the concept of Fast-Path Channels in a multiserver system called NewtOS. These channels are used to improve the communication between OS components that are running on dedicated cores. Moreover, they demonstrated this technique on a LwIP network stack which was split into small servers to increase dependability.

The Fast-Path Channels proved that multiservers can be fast and reliable. Thus, it answered the above research question. However, since it splits OS components in very small servers to improve reliability and it dedicates a core for each server to increase performance, this results in a large number of cores dedicated to the operating system. With the current implementation, these cores cannot be used by other applications, not even when they are idle. This is a waste of cores because you do not always need them. In this thesis we focus on improving the Fast-Path Channels to reduce the number of wasted cores. Our approach consolidates the servers on fewer cores when the network is idle or under low load, and expands back to more cores when the traffic is high. This way we can still deliver good performance and return unused (or less used) cores to user applications.

We think that in practice we do not always need powerful cores to run OS components, especially for small components like our network stack servers. Specifically, we expect to see in the near future asymmetric multicore processors with a few powerful cores (like today's processors) and tens or hundreds of low power cores[10] on the same processor die. For that reason, it might be better to off-load small OS servers on multiple low power cores. In this work we focus on analysing and evaluating how multiserver systems can benefit from such processors to make the system more efficient in terms of CPU usage, with less impact on performance and without affecting reliability.

An observation from the recent development of multicore processors is that hardware vendors focus on heterogeneous multicore processors with potential for power reduction. We have seen this type of processors in embedded systems where power usage matters the most. For example NVIDIA Tegra 3, a quad core system on a chip, contains a fifth companion low power core, much smaller than the other four and with fewer features. The companion core was designed to be used by the operating system when the device is mostly idle, to shut down the other cores. TI OMAP5 has two low power and two powerful cores which can be used by applications or OS depending on the workload. All these cores share the same Instruction Set Architecture (ISA), which means that a program compiled for that architecture, can run on any of those cores. This type of architecture is called big.LITTLE [5], and in the future we expect to see it not only on embedded systems, but also on workstation systems.

[1] AIX is an IBM propriatary UNIX OS

Our contribution to this work is two fold.

- First, we analyze OS components that require dedicated cores for performance reasons, to find what kind of processing power they need to handle different workloads, and to offload them to low power cores based on these workloads. In the future we expect to see processors with a large number of low power and specialized cores which can be used by OS components that do not require complex features found in current processors, like out-of-order execution or branch prediction [11]. This way we free the powerful cores and make them available for applications that can harness their real power. We base our entire analysis on the NewtOS networking stack which already supports fast-path channels. However, we can apply the same principles to other OS components as well.
- Second, we extend the fast-path channels library to consolidate the new network stack to fewer cores when the required bandwidth is low, so that part of the unused cores can be reclaimed by other applications. When more bandwidth is needed, we expand the network stack to dedicated cores in a similar way virtual machines expand to more physical machines in the cloud model.

Because asymmetric multicore processors with a large number of low power cores are not available yet, we simulate them using currently available hardware and a few techniques to vary their power/performance, so that we can verify whether our expectations make sense.

- Frequency scaling: Each modern processor supports frequency/voltage scaling. We use this technique to scale down the frequency on some cores, to create a platform with few powerful cores and many low power cores. The simulation ins not accurate, because those scaled down cores still support sophisticated micro-architectural features, while low power cores may have a simpler pipeline [15]. However, due to their simplicity, our network stack servers, may not take advantage of those sophisticated features. Therefore, scaling down the frequency should provide us a good estimate on how the network stack would perform on an asymmetric platform.
- Hardware supported multithreading is a feature available on Sun Niagara and recent Intel processors, which offers additional sets of registers, allowing multiple threads to use different processor units (available on a core) at the same time. An advantage for the current implementation of our fast-path channels is that each set of registers also includes the registers used by the MONITOR/MWAIT instructions. Thus we can dedicate for each network server a hardware thread instead of a physical core. Hyperthreading also reduces the number of CPU stalls due to frequent memory accesses, which may be beneficial for our network servers.
- Turbo Boost is another hardware capability which powers down idle cores and redistributes the saved energy to remaining cores in order to boost their performance (by increasing their voltage and frequency). We use this technique to increase the frequency on application cores while the cores dedicated to the network stack are idle.

This thesis is structured as follows. In Chapter 2 we give a brief introduction to Fast-Path Channels and how these are used in a fast and reliable networking stack. The implementation details of the frequency scaling driver and channel support for consolidation are explained in Chapter 3. Chapter 4 presents our performance evaluation. Chapter 5 compares our solution to other related projects and Chapter 6 concludes.

Chapter 2

Background

As we are basing this work on a previous implementation of Fast-Path Channels, we are going to describe in this chapter the main concepts underlying their implementation, as well as their advantages. To understand these concepts, we also depict a scenario where channels are used to improve the performance of the network stack, in a multiserver operating system. We are going to further use this scenario in our evaluation process.

2.1 What are Fast-Path Channels?

In micro-kernel based operating systems, like multiservers, traditional IPC (Inter-Process Communication) imply kernel involvement every time a process wants to communicate with another process. Mode and context switches come with an overhead cost for entering and leaving the kernel. Every trap in the kernel pollutes caches and branch predictors, degrading system performance, especially for performance critical OS components like the network stack or IO system, where a lot of messages are exchanged. On multicore systems, when the source and destination processes run on different cores, the kernel trap is followed by an interprocessor interrupt for inter-core communication, which adds to the total overhead per IPC. Moreover, the entire synchronous communications, as used in most multiservers, introduces unnecessary waiting and reduces the amount of parallelism. This scenario and the overhead are visualized in Figure 2.1a, where one process(A) sends synchronous messages to another process(B) and both of them switch between user mode and kernel mode for every exchanged message. This overhead sums up with each message and renders the entire system inefficient in terms of performance.

Fast-Path Channels, as described in [7], are an improvement over the traditional IPC, which makes kernel involvement unnecessary, and increases the communication speed between performance critical OS components. This means that all messages are exchanged directly from user mode as shown in Figure 2.1b, and the overhead associated with each message is removed. Besides this change, Fast-Path Channels use asynchronous messages to avoid unnecessary waiting. This improves performance, allowing the sender to do more work after sending a message, instead of waiting for a response. Asynchronous communication also improves reliability, by preventing processes from

blocking on faulty OS components. In Figure 2.1b we can observe that asynchronous user mode communication significantly increases the IPC throughput between the two processes. Compared to the traditional IPC represented in Figure 2.1a, with channels, the sender is able to transmit noticeably more data in the same amount of time. To squeeze more performance from this design, the two processes are scheduled on dedicated cores so that they can run uninterrupted. Since the two processes run on dedicated cores, when there is no more data to transmit/receive, the cores are put into sleep mode to preserve power. This is represented by the switch to kernel mode in the right-hand side of Figure 2.1b.

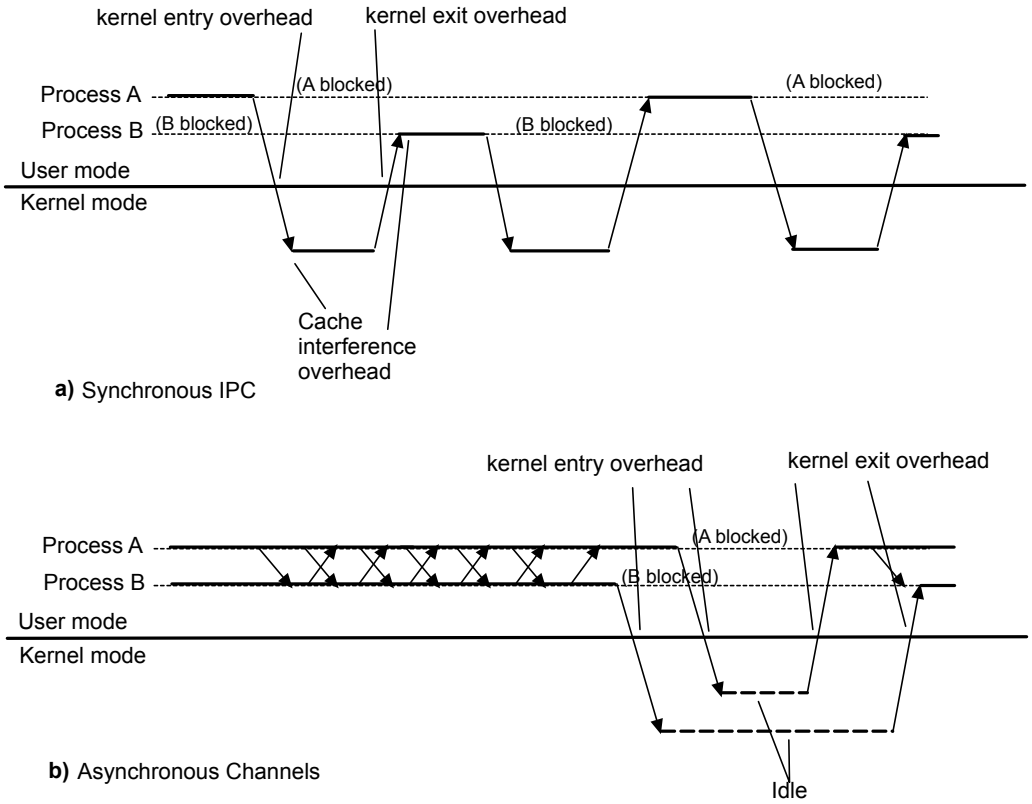

Figure 2.1: Overhead - Synchronous IPC vs. Fast-Path Channels

At their core, channels use shared memory queues to send requests, shared memory pools for passing large data and a database for tracking unreplied requests. For example, a sender places the data in a memory pool and a request in the requests queue. Each request represents a fixed size message which points to the data in the memory pool. At the other end, the receiver does busy waiting polling on the requests queue, reads each request and then accesses the data associated with the request, from the memory pool. All these shared memory regions are packed in a single and unidirectional channel, which is established in the initialization phase of an OS component. Each channel is established using synchronous IPC to setup the shared memory regions. For two-way communication, two channels are established. The second channel is used by the receiver to reply back to the sender, by placing a reply message in the requests queue of that channel. Thus, both end-point processes act as a sender for one channel and as a

receiver for the other one. When the sender receives a reply it uses the request database to find the corresponding sent message, which is then removed from the queue.

The busy waiting on the requests queue is deliberately done to avoid kernel involvement. However, when the queue is empty, instead of busy waiting forever, the implementation uses a more effective solution based on the x86 MONITOR/MWAIT instructions, preventing unnecessary waste of power. MWAIT causes a core to go to sleep while MONITOR tracks a specific memory region and wakes up to core when that memory region gets modified. To use this feature, each channel adds an extra shared memory buffer, which is monitored by the receiver and changed by the sender for waking up the receiver when new requests are ready to be processed. Before the MWAIT instruction is called, the receiver does a few more polls after the request queue becomes empty. This ensures the receiver that the sender finished to transmit all messages, and can go to sleep. For every message sent, the sender increments a counter in the shared buffer, called *notification area*, to guarantee that the receiver wakes up and becomes ready to process the requests.

The entire API for managing channels and all the functionality explained above, is packed in a small and portable library which can be used by any OS component.

2.2 High Throughput Networking Stack

As a proof of the Fast-Path Channels concept, T. Hruby et. al. [7] designed a new networking stack, which is based on LwIP, to demonstrate that multiserves can be made faster by using a better design for the IPC mechanism. The channels allowed them to break the network stack further into small servers (one for each layer), instead of one or few bigger servers for the entire stack. Multiple isolated components are good for fault tolerance. Also, with multicore systems and asynchronous messages, each stack layer can do more work in parallel without waiting for other layers.

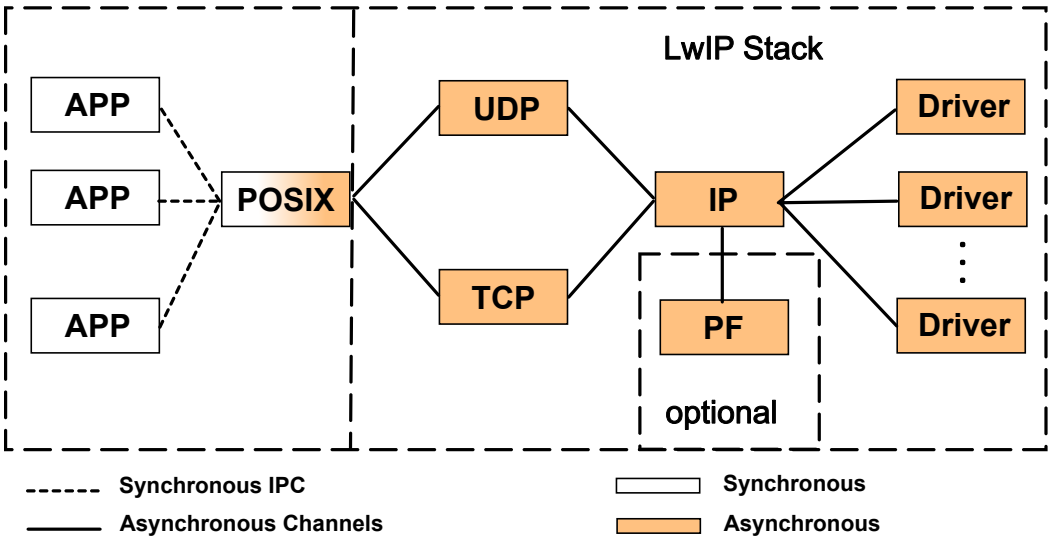

Figure 2.2: LwIP Architecture with Fast-Path Channels

The overall architecture is shown in Figure 2.2. Each shaded box represents a network server and uses a pair of two channels (for both directions) represented by the continuous line interconnections. The POSIX server translates synchronous calls to asynchronous messages, between user applications and the network stack. It implements network library calls (like `socket()` for opening a new socket, `read()` and `write()` for transferring data over the network, etc.) used by applications to send and receive data over the network.

Chapter 3

Implementation

In this chapter we present the implementation details, the project components and how they relate to each other and the rest of the operating system. We start by explaining the ACPI subsystem to understand how we are going to use it in our frequency scaling driver. Further, we describe the frequency scaling driver itself and how we used it to simulate a heterogeneous multicore platform. Finally, we present the changes that we made in the channels library to allow server consolidation.

The development platform is the same as the testing platform, which consists of two x86 PCs, running a multiserver operating system called NewtOS which is a fast and very reliable operating system with the ability to recover and replace its faulty components without disrupting the normal operation.

As explained in the introduction chapter, we use frequency scaling to simulate a heterogeneous multicore platform. Since our multiserver operating system did not support frequency scaling, we implemented a driver which provides this feature. Since each processor has different frequency scaling capabilities, we describe a solution that supports many X86 processors, by detecting at runtime the processor type and performing the proper initialization for that processor. To achieve this functionality, the driver has to interact with other OS components, which makes it more complex from the implementation point of view. The overall architecture is shown in Figure 3.1, where the lines between each server and driver show the interactions between them.

The DS server is a DataStore which offers a publish-subscribe service. We use this server to announce when a new driver is available. For example, the frequency scaling driver can be initialized only if the ACPI driver is up and running. To achieve this, each driver/server publishes its name when it starts. The frequency scaling driver subscribes for the ACPI driver and the scheduler subscribes for the frequency scaling driver. Once the ACPI driver is online, the frequency scaling driver will request all performance stats for each CPU. After the frequency scaling driver this is initialized, the scheduler can send commands which the driver will translate into kernel calls. The kernel will forward the commands through SMP code to the target CPU. In the next sections we explain each component, that we changed, in detail.

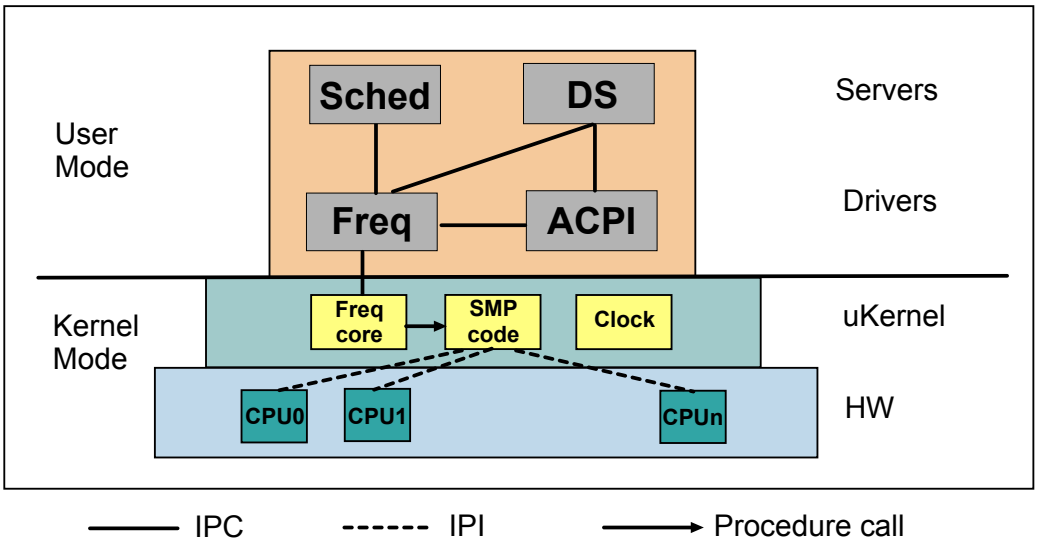

Figure 3.1: Frequency scaling driver - high level overview

3.1 ACPI driver

Our frequency scaling driver uses the ACPI standard to detect processor capabilities that different vendors integrate in their hardware. Since the ACPI is very complex, we are going to explain in this section all its terms and components that we are using in our frequency scaling driver.

The Advanced Configuration and Power Interface (ACPI) is a specification which provides an open standard for device configuration and power management by the operating system. It describes hardware interfaces that are abstract enough to allow flexible hardware implementations and concrete enough to allow OS to use these interfaces for hardware discovery, configuration, power management and monitoring. The standard improves existing power and configuration standards and transition towards entirely ACPI-compliant hardware which brings the power management under the control of the operating system, as opposed to previous BIOS-central systems, found on legacy systems, where power management policy decisions were made by the platform hardware/firmware shipped with the system.

Any ACPI-enabled operating system has to implement the ACPI specifications. However, the standard is so complex that it can lead to a lengthy and difficult implementation in the operating system. This issue is addressed by the ACPI Component Architecture (ACPICA)[2], an open-source and OS-independent implementation of the ACPI standard. The ACPICA implementation is delivered in source code form (usually ANSI C) and can be hosted by any OS just by writing a small and simple translation layer between ACPICA and the operating system. This layer is called the OS Services Layer and provides to the ACPICA, access to OS specific resources like memory allocation and mutual exclusion mechanisms. Therefore, the ACPICA subsystem consists of two major software components:

- The ACPICA Core Subsystem which provides the fundamental ACPI services (through Acpi* function calls), that are independent of any operating system.
- The OS Service Layer provides the conversion layer that interfaces the ACPICA Core Subsystem (through AcpiOs* function calls) to a particular host operating system.

These two components are integrated into the host operating system as an ACPI driver. The driver is then used by other operating system components which form the OS-directed configuration and Power Management (OSPM) subsystem. The OSPM is responsible for device configuration during system initialization, and implements different policies for power management and system monitoring. Figure 3.2 shows the general ACPI software and hardware components relevant to the operating system and how they relate to each other.

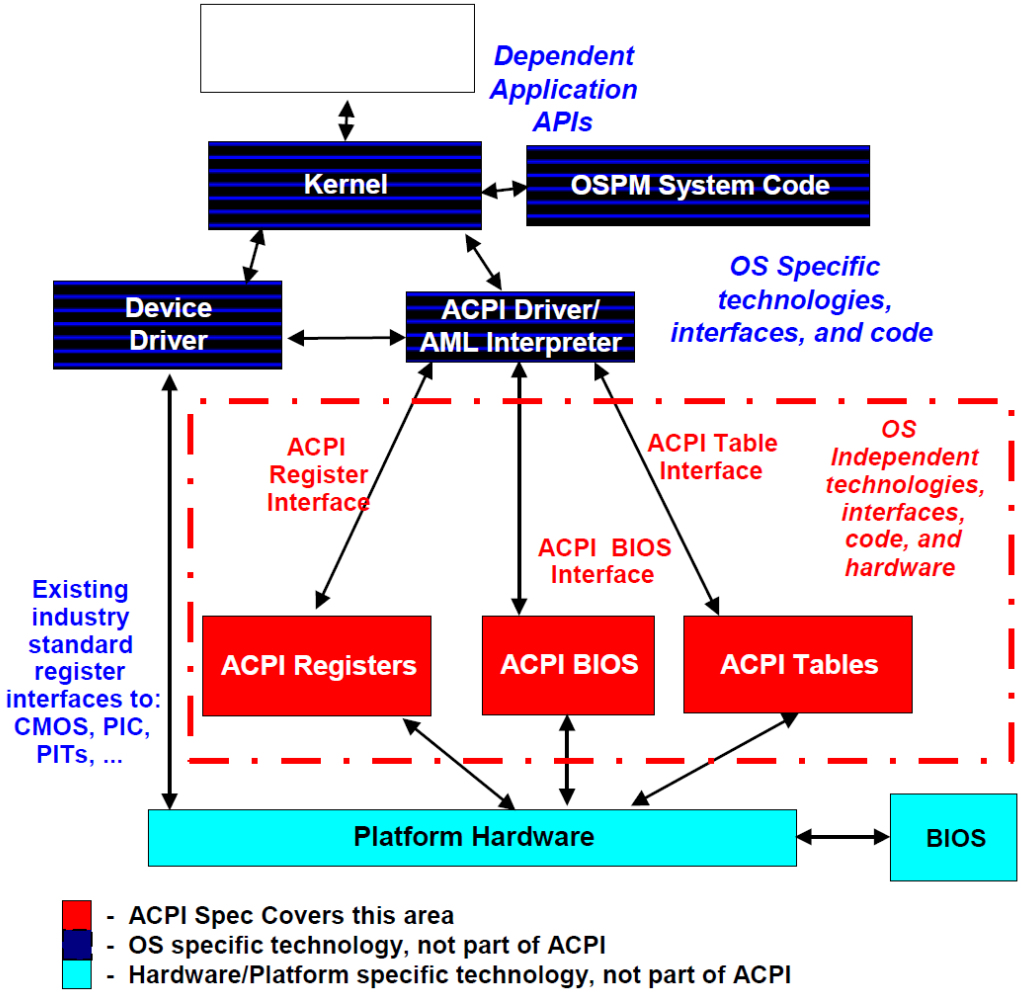

Figure 3.2: ACPICA General Architecture - taken from ACPI manual

To give hardware vendors flexibility in choosing their implementation, ACPI uses System Description Tables to describe system information, features, and methods for con-

trolling those features. These tables list devices in the system plus their capabilities. The ACPI System Description Tables represent the core of the ACPI standard, and contain *Definition Blocks* in form of ACPI Machine Language (AML) byte-code which describes and implements methods which allows the OS to control the hardware. The byte-code is interpreted by the host operating system, using the ACPI Machine Language (AML) interpreter, which is a component of the ACPICA Core Subsystem. A Definition Block is first written into a human readable language called ACPI Source Language (ASL), which is then compiled into AML images and shipped with the hardware. Each Definition Block starts with an identical header and resides into a namespace scope, which we will introduce and explain later in this section.

The platform firmware/BIOS is responsible to setup a Root System Descriptor Pointer (RSDP) in the system's physical memory address space. The operating system uses this pointer to find and map all standard Description Tables into the kernel's virtual address space. These Description Tables are stored in memory as shown in Figure 3.3.

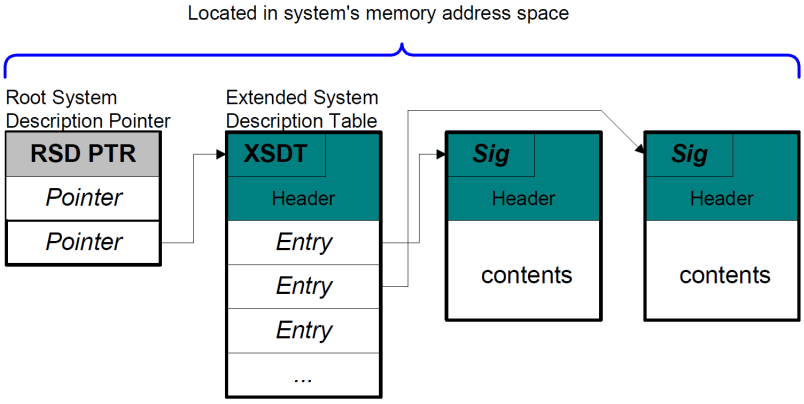

Figure 3.3: Root System Description Pointer - taken from Intel ACPICA User Guide

The RSDP structure stores the address of the Root System Description Table (RSDT) and the Extended System Description Table (XSDT) that contain references to other description tables. All tables start with identical headers. Besides these two tables, the ACPI standard offers the following tables:

- Fixed ACPI Description Table (FADT) - defines various fixed hardware ACPI information vital to an ACPI-compatible OS, such as base address for different hardware register.
- Firmware ACPI Control Structure (FACS) - contains read/write structures exported by the BIOS
- Differentiated System Description Table (DSDT) - this is part of the system fixed system description which lists the available hardware components on the main board.
- Multiple APIC Description Table (MADT) - describes the Intel Advanced Programmable Interrupt Controller(APIC) and all interrupts for the entire system.

- Secondary System Description Table (SSDT) - the system firmware can export multiple optional SSDT tables which are a continuation of the DSDT that describe all capabilities for different hardware components.

In the initialization phase, the ACPICA Core Subsystem recursively loads all tables (except SSDT tables) into an ACPI Namespace. For all Definition Blocks in all description tables, the system maintains a single hierarchical ACPI namesapce. Each node in the hierarchical structure is referred as an ACPI object which corresponds to a definition block for each device. An ACPI object is identified by a unique name which determines its position in the hierarchy (also known as object scope). The root node starts with the '\' character and lists all hardware components that interconnect the entire system (e.g. the main bus). Also, an ACPI object has a type that defines the format of the object. Among the most important types we mention:

- Object - this is a generic type which contains a header common to all types. Any type can be converted to an Object.
- Processor - defines a processor and all its capabilities.
- Package - contains a vector of other objects, which describe different capabilities.

For example, a processor is represented by the following node expressed in ASL code. Each package at the end of the code can point to other objects which identify different processor features and capabilities (e.g. power states, etc.)

```
Processor (
    \_SB.CPU0, // Processor Name
    1,         // ACPI Processor number
    0x120,     // PBlk system IO address
    6 )        // PBlkLen
{
    Package {
        Resource[0] // Package
        ...
        Resource[n] // Package
    }
}
```

Listing 3.1: ACPI Processor node

The operating system can access an object by evaluating (interpreting the AML code) the definition block, using the API provided by the ACPICA Core Subsystem. As a result, the operating system will get a C structure which is a one-to-one translation from the ASL code which describes that object. The API also provides functions to walk through the entire namespace to search objects by different criteria and to return their evaluated result.

Our multiserver operating system already has an ACPI driver which uses the ACPICA implementation and extends it with an OS Service Layer and a small part of the OSPM subsystem. The OSPM includes only the ACPI interface for a PCI driver and does not include any implementation for power management or frequency scaling. Next we explain how we extended the OSPM with these features.

In order to control the frequency, the OSPM has to know the supported frequencies for each core. This information is stored in an optional SSDT table which is not loaded by default into the ACPI namespace. We extended the OSPM to instruct the ACPI driver to load the corresponding SSDTs by specifying the Processor Driver Capabilities (_PDC). The _PDC is an ACPI object which calls an AML method to inform the hardware platform about the level of processor power management support provided by OSPM. The level value is passed to the function which evaluates the _PDC object. In our case, we have to inform the hardware that our OSPM implementation supports performance management. Once the ACPI driver loads the SSDT tables with performance capabilities, we have to read the following objects:

- Performance Control (_PCT) - contains the control register and status register, used for changing and reading the current frequency.
- Performance Supported States (_PSS) - a list with all supported frequencies and the corresponding power usage.
- Performance Present Capabilities (_PPC) - specify which frequencies from the _PSS list are supported at the current time. This can change when the processor supports Turbo Boost, depending on the number of powered down cores.
- P-State Dependency (_PSD) - this object assigns a domain number for each core. The power management unit on the processor requires that all cores in the same domain to be scaled at the same time, with the same frequency. Each domain has a coordinator which can be a software (the OSPM, if the BIOS supports software coordinators) or a hardware coordinator. A coordinator is responsible for switching the frequency on all cores in the same domain. For example, with a hardware coordinator, the OSPM can change the frequency for only one core and the hardware takes care of the rest.

All the above objects are within the namespace of each processor (core), like in the following ASL code example with 3 supported frequencies.

```
Processor (
    \_SB.CPU0, // Processor Name
    1,         // ACPI Processor number
    0x120,     // PBlk system IO address
    6 )        // PBlkLen
{
    Name(_PCT, Package () // Performance Control object
    {
        ResourceTemplate(){Register(FFixedHW, 0, 0, 0)}, // PERF_CTRL
        ResourceTemplate(){Register(FFixedHW, 0, 0, 0)}  // PERF_STATUS
    } // End of _PCT object

    Name (_PSS, Package()
    {         // freq, pow,   lat,      ctrl, stat
        Package(){650, 21500, 500, 300, 0x00, 0x08}, // (P0)
        Package(){600, 21500, 500, 300, 0x01, 0x05}, // (P1)
        Package(){500, 21500, 500, 300, 0x02, 0x06}, // (P2)
    }

    Method (_PPC, 0)  // Performance Present Capabilities method
    {
```

```
        If (\_SB.DOCK)
        {
            Return(0) // All _PSS states available (650, 600, 500).
        }
        If (\_SB.AC)
        {
            Return(1) // States 1 and 2 available (600, 500).
        }
        Else
        {
            Return(2) // State 2 available (500).
        }
    } // End of _PPC method

    Name(_PSD, Package () // P-State Dependency
    {          //num,rev,dom, coord,procs
        Package(){1, 0x1, 0, HW, 1}
        Package(){2, 0x1, 0, SW, 2}
        Package(){2, 0x1, 0, SW, 2}
    }
} End of processor object list
```

Listing 3.2: ACPI Processor node

After the ACPI driver is initialized, we walk through the namespace starting from the root node, and for each processor that we find on the main bus, we get the data from the objects described above. We store the data on a per processor structure and we provide an API which is used by other drivers to request the data from the ACPI driver. When we store the data we have to convert the processor ID found in the ACPI object with the ID used by the operating system. The OS uses the ID advertised by the APIC interrupt controller. To make the conversion between the two IDs we have to look for each processor into the ACPI MADT (Multiple APIC Description Table). This table describes the interrupts for the entire system, but also contains the processor ID.

3.2 P-states and frequency scaling driver

Processor P-states represent the capability of a processor to change its performance at runtime. Each core in a processor can have many P-states and each P-state defines a clock core speed (the frequency) and an operational voltage. Besides these characteristics, each P-state has a switch latency which is the time it takes to switch to that state, and during which the processor does not execute anything. Like C-states, the switch latency can be different between P-states. However, the P-state latencies are lower that C-state latencies.

The P-states are changed by our frequency scaling driver, which consists of three components:

- A generic part - which implements an API to be used by the scheduler. It also defines a set of handlers which are implemented by a CPU specific driver, and a small interface to register new handlers from a CPU specific driver.

- CPU specific driver - is responsible for hardware initialization and actual scaling. Currently, we have two drivers, for Intel processors and AMD Opteron and Phenom processors. Both drivers interact with the ACPI driver to get all supported frequencies and to build a frequency table for easier access based on a specified frequency. Each driver implements a probe method for testing if the current hardware is compatible with the driver. This way, even if we have multiple drivers, the generic driver will initialize and install only the specialized driver that works with the running hardware.
- Kernel driver - the frequency is changed only after a model-specific register (MSR) is written. This register cannot be accessed by a user mode process. It can only be accessed from the privileged mode (from the kernel mode). For that reason, we need to have a small part of the driver in the kernel, which receives CPU frequency scaling commands from the scaling driver. A command specifies the operation (read/write MSR or change kernel clock and cpuinfo data), the MSR address and the value to be written. MSRs are specific to each core and only the target core can change the frequency. As we cannot make sure that the kernel call for changing the frequency is issued on the target core, the current core must send an inter-processor interrupt to tell the target core to change its frequency. Before the interrupt is sent, the sender updates per core pointers to pass to the receiver the command and a function to execute. This is a messaging mechanism used between cores to execute a specific function on a target core. In our case we pass a function which writes a value to a MSR.

3.3 Core-expansion/consolidation for OS Components

With Fast-Path Channels, the message notification send is just a write in a shared buffer (the notify area). If the receiver is sleeping, then the processor resumes its work and the receiver wakes up. On the other hand, if the receiver is running on a shared core, it cannot use the MWAIT mechanism anymore, because otherwise it would halt the entire CPU and block all other processes. Instead, it has to use a blocking receive. Also, the sender has to use a notify call to wake up the receiver from the blocking receive. If the receiver is moved from a dedicated core to a shared core at runtime, the two processes do not know which notification/receive method to use. They could use blocking receive and notify all the time, but this would be against the purpose of channels, to avoid kernel involvement as much as possible. For the receiving part, we can solve this problem because receive and MWAIT operations are both implemented as kernel calls. In this case the kernel keeps a counter for each CPU with the number of processes scheduled on that core. Thus, when the counter is 1, the kernel will switch to MWAIT and when the counter is higher than 1, it can use blocking receive. However, the sender does not know when to use notify. In our solution the scheduler (since it is the one who dedicates cores) informs the sender when to use notify. The scheduler keeps a shared bitmap which can tell for each process in the system whether is running on a dedicated core or not. In our implementation, the shared bitmap is a channel buffer exported by the scheduler as read-only. Other processes then map this channel and check the bitmap every time they send a notification to see if the destination is

running on a dedicated core or not. We integrated this check in the channels library so that there is no need to change to servers' code. To avoid possible race conditions between the scheduler and the notification operation, we make an extra `notify()` call every time the scheduler moves a process to a different core.

Chapter 4

Evaluation

4.1 Experimental Setup

Our test environment consists of two commodity machines each having a large number of cores. To emulate a heterogeneous platform, we used the method explained in the previous chapter to scale down the cores to lower frequencies. We conducted experiments on each machines separately to test different hardware capabilities. The used machines are the following:

AMD Opteron: The first machine is a server with an AMD Opteron 6168 twelve-core processor running at maximum frequency of 1900 Mhz and which can be scaled down to 800 Mhz, with 8GB RAM and 5 x 1Gbit Intel network cards, limited by the number of PCI express slots on the main board. The main advantage of this machine is that we can scale each core individually, which allows us to simulate a platform with asymmetric cores that support different frequencies.

Intel Xeon: The second machine is a server based on two Intel Xeon E5500 quad-core processors running at maximum frequency of 2267 Mhz, having 16 hardware threads (two threads per core), 8GB RAM and 4 x 1Gbit Intel network cards, also limited by the number of PCI express slots. Compared to the previous machine, this machine does not support individual core scaling. Instead, the entire socket has to be scaled to the same frequency. Fortunately, we have two sockets and therefore, we can simulate a big.LITTLE system. This machine however, gives us a new feature which is not supported by the first machine, hyperthreading. With hyperthreading we can use fewer physical cores and hide CPU stalls due to memory accesses.

As opposed to the first machine, the second server is more powerful, has only 4 network cards and can be scaled only to 1600 Mhz. In order to saturate the 4 Gbit cards and the entire networking stack (for testing under heavy load) we had to introduce some overhead by adding the Packet Filter (PF) component. The PF was configured to behave like a stateful firewall which keeps track of all network connections traveling across it. The firewall is configured with static rules to distinguish legitimate packets from those that belong to prohibited connections. For each connection established from behind the firewall, a dynamic rule is created automatically to allow all replies to pass

the firewall. All these rules (static and dynamic) are stored in a red black tree (or a hash table), which is searched every time a packet passes through the network stack. Therefore, a large number of rules can degrade the network performance. The more connections you initiate, the more rules will be added and thus the filtering overhead increases.

On both machines we were able to saturate the network cards with 4 TCP streams per 1Gbit card. At the other end of each stream we used a Linux machine connected directly to our test boxes. We implemented a small TCP client to stream 16 KB of data from our machines to an iperf server running on the Linux system. Iperf is a network testing tool used to measure network throughput with UDP or TCP data streams. The next sections show the tests we ran on each machine and discuss their results.

4.2 Low Power Cores on AMD Opteron

To find the performance requirements for each network server, we start by scaling the frequency for each of them individually. This way, we hope to find which components require powerful cores and which can run on low power cores. Since the IP, driver and the POSIX components are simpler than TCP, we expect to see comparable results between runs with different frequencies.

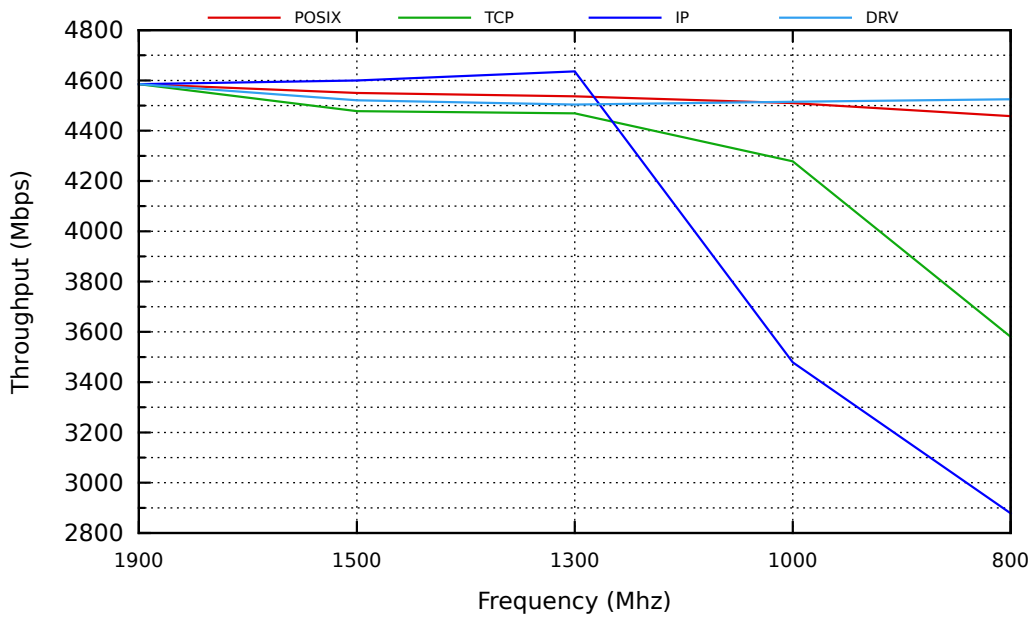

Figure 4.1: AMD - Throughput vs. Frequency

In Figure 4.1 we represent the network throughput when each server runs at different frequencies. When all components run at high speeds, we observe that the slowest component is TCP, which makes sense because the TCP server has the most work, since it is a stateful component. It has to track all connections, keep all unacknowledged packets in memory for retransmits, perform congestion control and other operations

which make TCP the most complex protocol in the entire TCP/IP stack. Thus, the TCP server bounds the maximum network performance. As expected, for most cases, scaling down the frequency reduces the achieved throughput. However, we observe that scaling the IP server down to 1300 Mhz, the throughput increases. To understand this behaviour we measure CPU usage for each server with IP running at different frequencies. The goal of the experiment is to observe how a scaled IP server influences the other components and the entire network performance. The results are shown in Figure 4.2.

On the left-hand side of the CPU-usage graph, when all servers run at full speed (1900 Mhz), the TCP and IP servers (the green and blue bars) experience some IDLE times. This is because the IP is faster than TCP, has less work to do and it goes to IDLE more often when it finishes its work. Because of the MWAIT operations, when there are no requests in the queue, the processor enters IDLE (sleep) mode by switching between different power states (C states). The processor does a switch between C0 (the operating state) and a higher C state (e.g. C1 or C7). For each state, the CPU turns off different components (like internal and external clocks, caches, bus interface unit) and the CPU state is saved in memory. Higher states means more power savings but also longer latency when returning to C0 [13], because the CPU has to power on all disabled components and to restore its state from memory. This latency adds up every time when the IP server enters IDLE mode. Thus, because of this latency, the IP is not able to fully load TCP and to deliver maximum throughput. If we decrease the frequency it becomes slower and the idle time is decreasing for both TCP and IP. At some point it does not go to idle any more, thus avoiding the mode switch overhead and the idling latency.

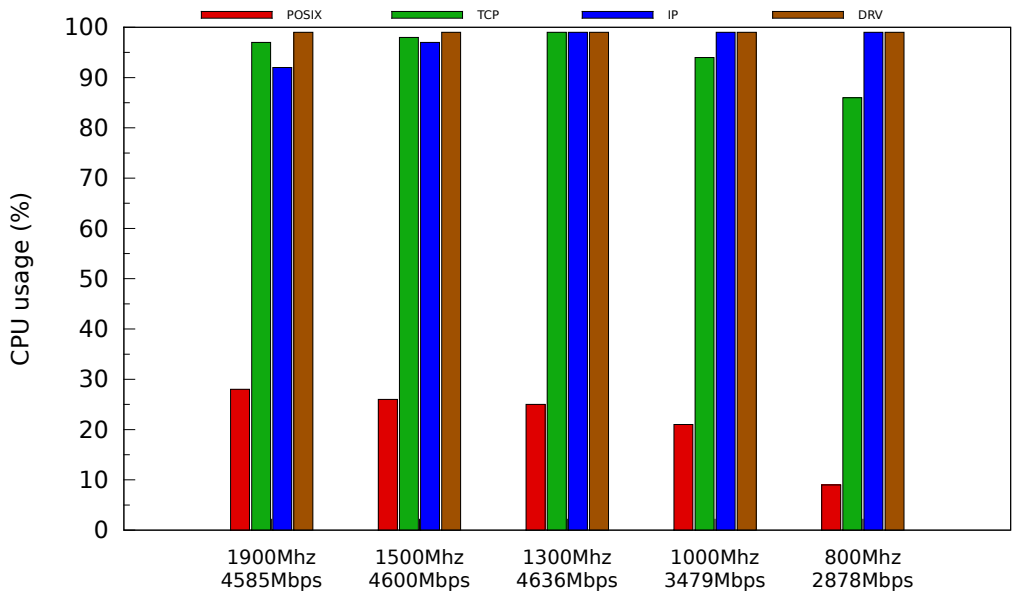

Figure 4.2: AMD - CPU usage for different IP frequencies

If we scale it further more (below 1300 Mhz), it becomes too slow to handle all requests and the throughput is also decreasing. We experienced the same behaviour on all tests that we ran on the Xeon machine. We explain this in more details for those tests, later in this chapter.

Earlier we saw that the driver and POSIX server can run at 800 MHz, and the IP server at 1300 Mhz. These are the optimal frequencies when each component was tested separately. However, scaling down a component may influence another one. For example, scaling the driver to 800 Mhz, may influence the IP server and 1300 Mhz may not be the optimal frequency anymore. In this case, to find the optimal configuration, we measure the network throughput when each server is running at different frequencies. The results are shown in Figure 4.3 where each configuration is represented in the legend as a tuple (POSIXxTCPxIPxDRIVER) identifying the operating frequency of each server. The red line in the figure represents the network throughput for the base configuration, when each server is running at full speed. An interesting configuration is when each server is running at the lowest frequency such that the entire network stack still delivers maximum throughput. This configuration is represented by the green line in the same figure.

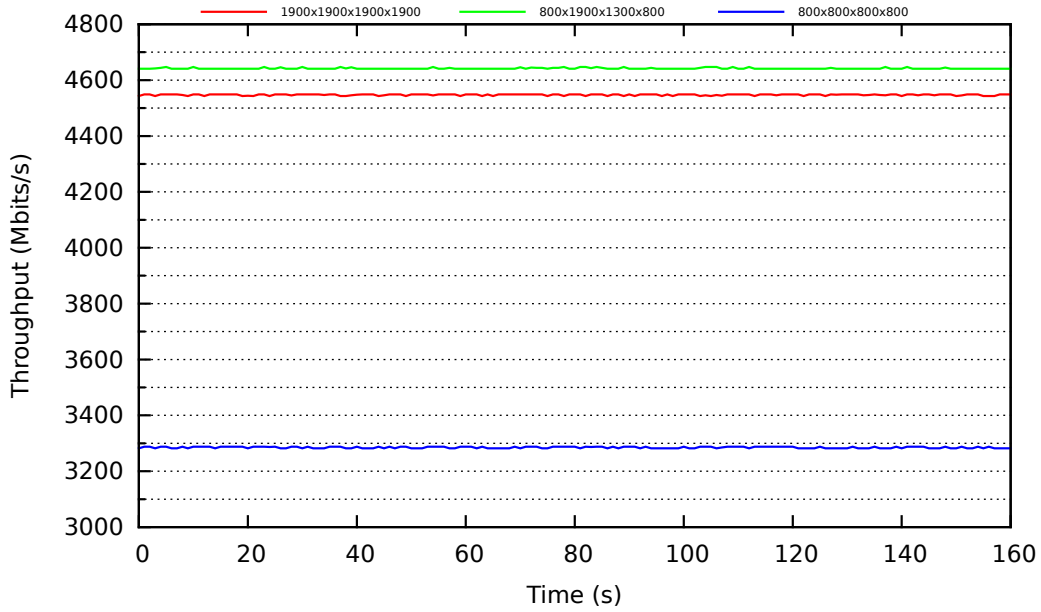

Figure 4.3: AMD - LwIP on low power cores

As we explained earlier for the IP server, choosing the right frequency to minimize the IDLE time for each server can provide better throughput. This is exactly what we expected to see with this configuration. The same result can be achieved if we increase the polling period after the request queue becomes empty. However, this would make the system inefficient in terms of power usage. Finally, the blue line is for all servers running at low speed or on low power cores. This is perfect for running the stack when less traffic is going through. As you can see, the stack is still able to deliver 3.3 Gbps, which is sufficient for many applications today.

4.3 TCP/IP Stack Consolidation on AMD Opteron

With our consolidation feature, we were able to test different parts of the network stack sharing one or more cores. A core can be shared in two ways: i) run other OS components (drivers) and a network server on the same core, by not dedicating another core for that server. ii) dedicate fewer cores to the network stack, and place multiple servers on the same core. We tested these scenarios to see how the network stack can be consolidated for different workloads to save more resources and conserve energy. The goal was to consolidate as many servers as possible on a single core. We started by consolidating all servers on the same core, but we experienced stalls which were caused by the TCP congestion control algorithm. This happens because all servers start to compete for CPU and there is no causality in when a process runs (due to the asynchronous nature of channels). TCP treats this behaviour as a congestion. Therefore, we were not able to get relevant results with the entire stack consolidated on a single core, but we successfully ran tests with three servers on a single core. The results are shown in Figure 4.4 where servers that share a core with other OS components are separated in the legend by a comma, while servers that share a dedicated core are separated by the '+' sign.

In this test, the green and blue lines represent POSIX and IP servers running on cores shared with other drivers or system servers. They were not running both on the same core. For the other plots in the graph, the network servers were consolidated on the same core, without sharing that core with other system precesses. The only exception is the red line which represents the base line (each server running on separate dedicated cores), for comparison reasons.

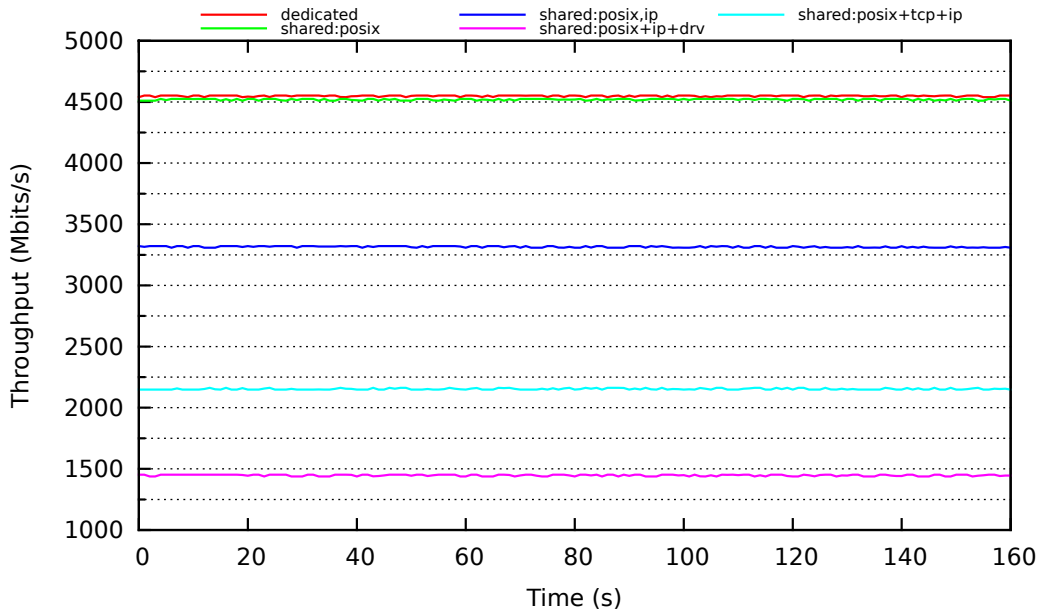

Figure 4.4: AMD - LwIP Consolidation

One of the conclusions that we can draw from this graph is that the POSIX server does

not need a dedicated core to deliver maximum throughput. This is because the POSIX server does less work and part of it uses synchronous communication which makes it to stay IDLE most of the time. For this reason, the POSIX server cannot take full advantage of the dedicated core and does not benefit from the channels. Thus, we can run the POSIX server like a normal system process, sharing a core with other drivers or OS components. From the last line, which corresponds to the lowest throughput in the graph, we can observe that sharing the driver server with other servers, results in a high performance drop. This is due the amount of polling done by the driver for each channel per network interface, which keeps the core busy all the time. This can also be observed from the previous CPU-usage graph where the driver keeps the CPU to 100% for any frequency. However, for low traffic conditions, a consolidated network stack which delivers 1Gbps throughput is still sufficient for many applications.

In the previous consolidation tests we used the same frequency for all servers. So far we have seen tests using only frequency scaling or consolidation. We can improve system efficiency by combining the two techniques. Hence, the next tests which show part on the stack consolidated on powerful cores while some of them are running on low power cores. Since we found the optimal frequencies for each server in the previous frequency scaling tests, we use the same values in these tests.

In Figure 4.5 the legend shows network stack components that run on shared but different cores separated by comma. Components that run on the same shared core separated by 'plus' sign and everything else that follows after semicolon run on dedicated cores. The numeric value specifies the frequency for each component. Those components, that have no frequency specified, run at maximum frequency.

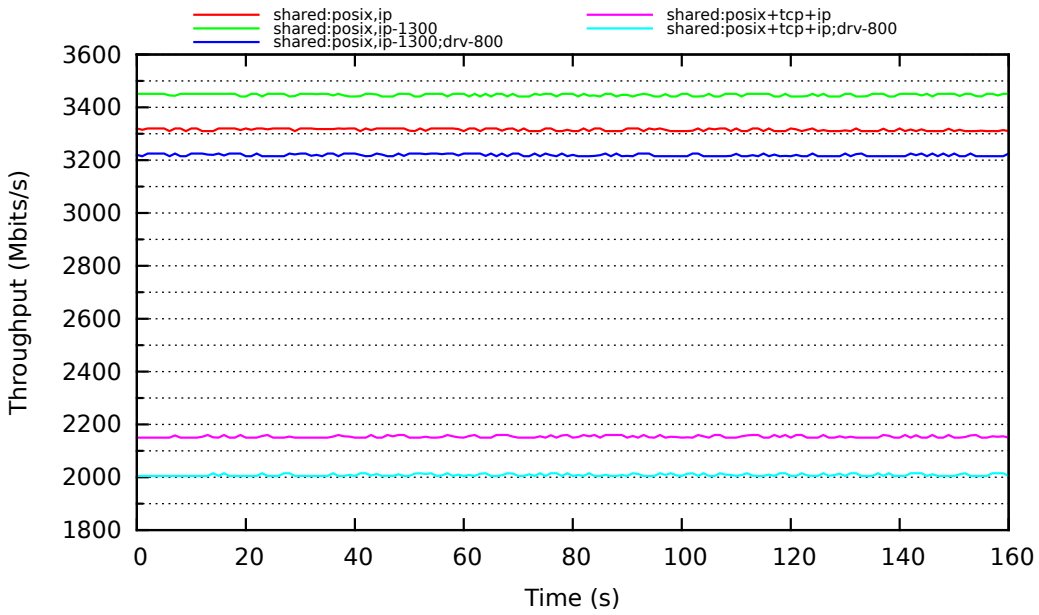

Figure 4.5: AMD - LwIP Consolidation on low power cores

4.4 Low Power Cores on Intel Xeon

On the second server we run the same experiments, but this time, as we explained at the beginning of the chapter, we introduce an extra component, the packet filter. With the new Packet Filter component, the slowest server becomes the IP server, as shown in Figure 4.6. The reason is that the IP server has a new channel to poll from, and for each packet, it has to make a request to PF and forward that packet to TCP only after a reply is received.

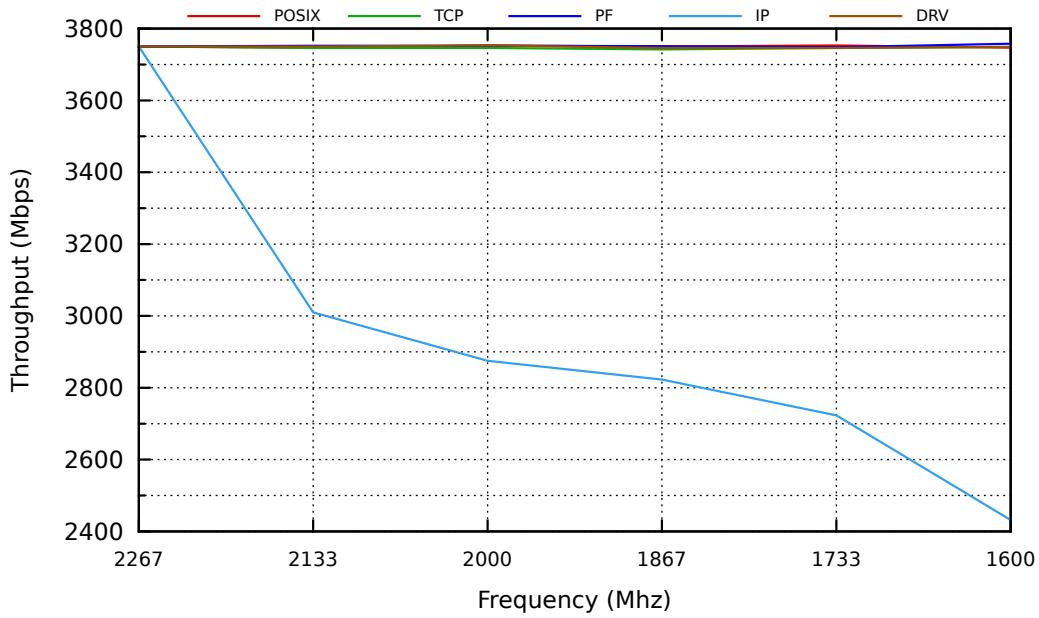

Figure 4.6: Intel - Throughput vs. Frequency

Running the other servers (individually) at different frequencies does not influence the network performance. Based on this observation, we would expect to see the same result if we run all servers but IP at low speeds. Figure 4.7 proves the contrary. The pink line which represents this configuration is significantly lower than the red line (the base line). Similar to previous tests (on AMD server), the five tuple, shown in legend, represents the frequencies for POSIX, TCP, PF, IP, DRIVER, in this order. The reason for this performance drop, is that both PF and TCP are slower and cannot keep up with the fast IP server, but not fast enough to handle both of them. Because IP is fast, it empties all channels and before it goes to IDLE is does a few more polls on empty channels. Before it finishes polling, the two servers send new messages and the process continues. Every time this happens, those few polls on empty queues add up to the total overhead and degrade the network performance.

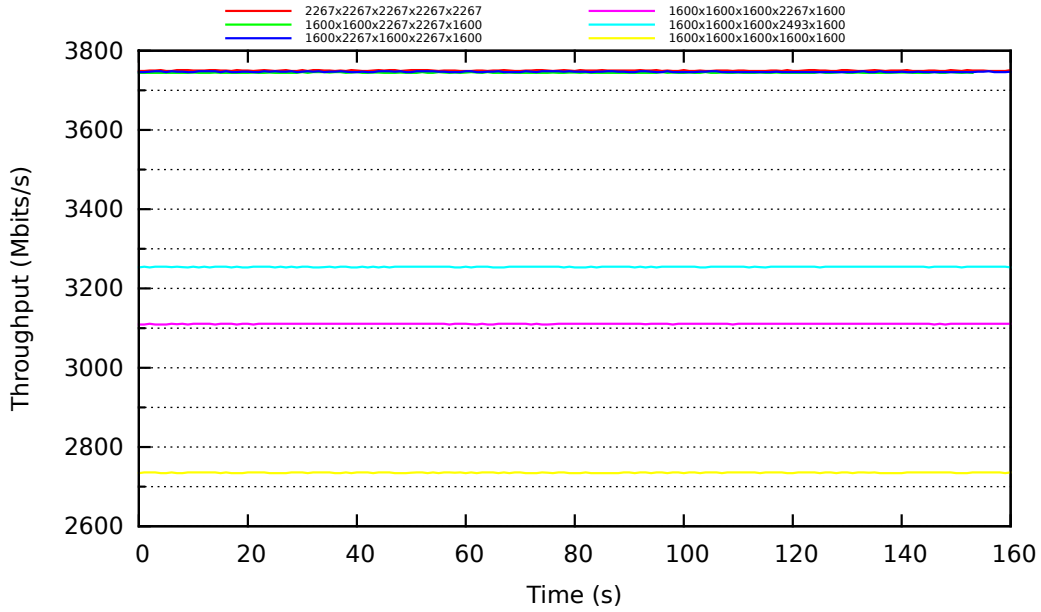

Figure 4.7: Intel - LwIP on low power cores

If we increase the frequency either for TCP or PF, we eliminate the empty channel polls and the performance returns to maximum. This is proved by the dark blue and green lines, which overlap with the base red line. Another solution is to increase the frequency for the IP server. 2267 Mhz is already the maximum frequency, but using Intel Turbo Boost we can increase the frequency even more. With two Intel Xeon quad-core processors we managed to create the following configuration.

- TCP, PF and DRIVER running on three dedicated cores.
- POSIX and all other OS components and user applications sharing a single core.
- all the above were placed on the same processor (package).
- IP running on a dedicated core from the second processor.
- the other three cores on the second processor were forced to IDLE.

While three cores were IDLE, we were able to boost the frequency for the IP server with 266 Mhz. The other servers were still running at minimum frequency (1600 Mhz). This test is represented by the light blue line (3350 Mbps). Even though this was not a big performance improvement (because of the low frequency boost), the solution proved to be feasible. On processors with more than 4 cores, the performance boost can be higher. Moreover, this approach can also be used for applications. We consolidate the network stack on fewer cores (when the traffic is low), power down the released cores and boost the application cores.

4.5 TCP/IP Stack Consolidation on Intel Xeon

In our final tests we took advantage of the two hardware threads per core, also called logical CPUs, supported by the Intel Xeon processors. From OS perspective, each logical core looks like a normal CPU. For this reason, in our previous tests we took care to schedule each server on different physical CPU core so that we can measure the maximum throughput. Figure 4.8 shows that running TCP and IP on two logical CPUs of the same physical core and the PF and the driver on other two logical CPUs, we get the same performance like in the previous tests. Basically we used only two physical cores instead of four.

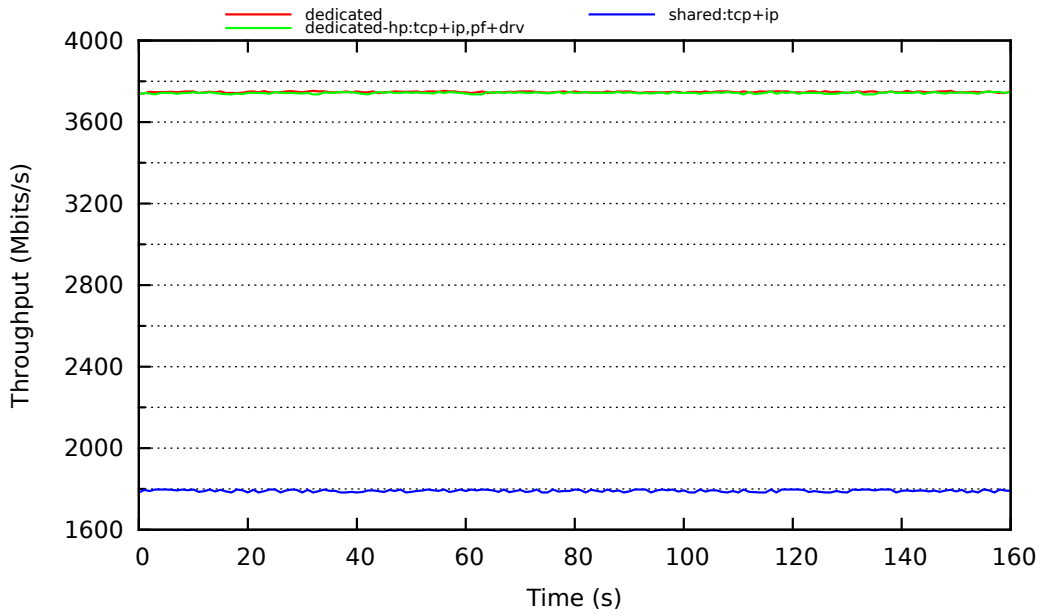

Figure 4.8: Intel - LwIP on hardware threads

This test corresponds with the green line in the figure, also marked as 'dedicated-hp' in the legend, while the red line is the base line marked with 'dedicated'. The blue line shows the network performance when running TCP and IP on the same physical core with hyperthreading disabled, while the other servers run on separate dedicated cores. With hyperthreading we can reduce the number of used resources while still delivering good performance.

4.6 Discussion

The tests prove our speculation made in the beginning, that we can run OS components on low power cores to improve the system efficiency. For performance we can expand to more powerful cores depending on the current workload and consolidate back when the load is low. Moreover, we show that in some situations, choosing less powerful cores for some components delivers better performance. This is a better solution than

increasing the amount of polling. Through experiments we also identified that IP and TCP servers require more power when the workload grows. On the other hand the network driver can run always on low power cores regardless of the current workload. We can schedule the POSIX server as a normal process without dedicating a separate core for it. Finally, hyperthreading delivered maximum performance even though we shared the same physical core. The main reason is because hyperthreading removes CPU stalls due to memory accesses, by scheduling another thread. This suggests that our network servers are more memory intensive than CPU intensive.

Based on these tests and results, we can implement a scheduler to dynamically select the best configuration for each workload. The scheduler would start with a consolidated configuration and when the workload is increasing it starts to expand components to lower cores and then to more powerful cores. In our case, the scheduler could increase the frequency first and then expand to scaled down cores and then increase the frequency for those cores. When the workload starts to decrease, we can apply the reversed process.

Designing such a dynamic scheduler implies some challenges, like (i) when should we decide to expand/consolidate and (ii) what components to choose. An idea is to extend the channels library to notify the scheduler when a queue is overloaded or underloaded (the queue size goes below a threshold). This gives an indication to the scheduler when the receiver is too slow or too fast and it can take actions to consolidate the receiver or move it to a faster core. For example, if the receiver is consolidated or running on a slow core and the sender is faster, the scheduler can decide to move the receiver on a faster core. If the receiver is already running on a fast core, the scheduler can move the sender on a slower core or consolidate it if it is on a slow core.

The scheduler would have to take into consideration other factors besides the queue size. In some situations it is not worth to expand or consolidate, especially when the workload changes frequently. In those cases the cost of switching cores can have a big impact on performance and power consumption, caused by a large number of cache and TLB misses after process migration. In heterogeneous systems the migration cost can be even higher. Fortunately, there is a lot of research which deal with this problem [4, 9, 12, 8].

A more advanced scheduler can also take advantage of Performance Monitoring Units (PMUs) that most modern CPU architectures (x86, ARM, MIPS) already support. A PMU contains a set of programmable hardware counters that can monitor different events (e.g. cache misses, branch predictions, bus cycles, etc.), without interfering with the normal execution of the CPU. An operating system can have per task and per CPU counters with event capabilities on top of them. With this mechanism we can trace hardware and software events like page faults, context switches, CPU migration, alignment faults, emulation faults and more, for each task and CPU. Weissel [19] shows in that these counters can also reveal power-specific characteristics of a thread. He suggests that different execution and memory access patterns have different power requirements and these requirements change with the frequency. The main idea is to use performance counters to track those patterns which have the best energy performance benefit from a reduction in clock speed. For example, the *memory requests per clock cycle* counter is used to monitor memory accesses, while the *instructions per clock cycle*

is used to monitor execution patterns. Therefore, a high the rate of memory requests will benefit more from a reduction in clock speed. For execution patterns, the rate of executed instructions has to be low to benefit from clock speed reduction. Using these counters the scheduler can determine the best trade-off between energy efficiency and performance.

Chapter 5

Related Work

Substantial prior work has proposed the use of heterogeneous processors to improve system performance and to reduce energy usage. Some of them are orthogonal to our design. For example, Saez et. al [15], use asymmetric multi-core processors to improve performance and power-efficiency of applications. They efficiently use asymmetric processors by core *specialization*, a method that splits the stream of instructions of an application in two types: *efficiency specialization* and *thread level parallelism specialization* (TLP). Parts of the application that have sequential code with instruction level parallelism use efficiency specialization and are assigned to powerful and complex cores so that they can take advantage of their out-of-order execution units. Other parts that are more I/O bound and do not take advantage of the complex features of the processor, or that contain parallel code, are assigned to multiple low power and simple cores using TLP specialization. Like us, they use commodity hardware to simulate a heterogeneous platform. However, on Xeon processors frequency scaling applies to an entire processor socket, rather than a single core. To overcome this limitation, they use clock modulation. With clock modulation, the CPU inserts periodic stalls by going to sleep. This method would not work with our channels, because as we saw from our tests, going to sleep too often degrades performance. Another difference is that they used Intel proprietary tools to defeature each core so that it behaves more like real small core. Unfortunately, we could not test this idea because those tools run only on Windows and Linux based operating systems and are designed for profiling user applications. In our case, these tools were not applicable since we are testing the components of an (unsupported) operating system.

The Click project [13] uses the same idea to build a power efficient software router because traditional hardware routers consume almost as much energy when they are idle or under low load, as they consume when are handling large traffic. It takes advantage of the heterogeneous multi-core processors to run the routing protocols on different core types, based on the workload. Because L. Niccolini et. al. tested Click only on Intel Xeon processors, they use clock modulation to simulate low power cores for the same reason like in the previous related work [15]. Like us, they also consolidate packet processing onto fewer cores when the network traffic is low.

Similar to our idea, the project described by J. C Modul et. al. [11], uses asymmetric multi-cores to improve operating system efficiency. It dedicates low power cores for OS

components, so that fast cores can be used by user applications. However, the major difference is that this design is implemented on top of a monolithic kernel (Linux) and they are running system calls on low power cores. Thus, when an application makes a systems call, the modes are switched between user and kernel, and the cores also have to be switched. Switching cores for every system call can have a significant negative impact on performance (with cache lines bouncing between cores), especially on system calls that execute fast. That is why the authors decided to switch the cores only for system calls that take longer to execute, so that the switching overhead is insignificant, compared to the energy efficiency gain. Also, the system call has to be long enough so that other applications can make use of the faster core. Their implementation decides if a system call is worth to switch the cores based on the size of the parameters. However, this kind of decision can be wrong. For example a non blocking `write()` system call can return faster, without writing everything, because the kernel buffers are full. In monolithic kernels, the OS components running in kernel threads can take advantage of dedicated cores. That is because they share the same address space and the switching overhead does not exist.

Gupta et. al. [6] investigate the limitations of using heterogeneous multi-core systems to gain energy efficiency. Besides the power usage of each core they also consider components that are shared between all cores. These components form the so called *Uncore*, which contains last level cache, memory controllers, power control logic, etc. These components consume power even when most cores are idle or powered down. With the integration of many types of cores on a single CPU die, the uncore component is growing and becomes an important factor in total system power.

Chameleon [14] uses dynamic processors to create a heterogeneous multi-core system. A dynamic processor consists of technologies like Turbo Boost or Core Fusion which can transfer the power from unused cores to other cores which need that power to increase their performance at runtime. In Chameleon the authors present an extension to the Linux operating system that allows fast reconfiguration of the processors to deliver better performance per watt. We also use this technique to show that applications can benefit from more power when the network stack is idle.

Chapter 6

Conclusions

6.1 Summary

This thesis presents how to make a fast and very reliable multiserver operating system more resource efficient by scheduling OS components on low power cores and by reducing the number of used cores for the these components, while still delivering good performance, using heterogeneous multi-core CPUs. We explained in detail the implementation of our frequency scaling driver that was used to simulate asymmetric cores with commodity hardware. We focused on extending the previous implementation that uses fast-path channels, to better utilize resources when the system is not loaded. To achieve this we used existing hardware technologies like frequency scaling, hyper threading and turbo boost. We also applied techniques found in clusters and cloud computing, like consolidation and expansion.

We ran more experiments on our modified version of the networking stack to analyze its behaviour on low power cores and to find a better way to utilize the cores based on the performance requirements of each component. Our tests showed that some operating system components don't need powerful and complex cores to deliver good performance. We even improved performance using lower CPU frequencies and we reduced the number of dedicated cores. Previously, we needed four cores only for the networking stack to run our new multiserver system. Now, with consolidation we can run it even on a laptop with dual core processor. In the future we expect to see processors with tens or hundreds of small and low power cores to accompany the existing powerful cores. Our design can take advantage of this type of hardware and extend the same approach to other parts of the system not only to the networking stack.

6.2 Directions for Future Work

All recent studies, including ours, focused on using single ISA heterogeneous multi-core architectures, to improve performance and energy efficiency. We think that using architectures with different instruction sets might deliver even better results. Some architectures are better specialized to specific workloads [10] and can deliver better

performance and utilize less power than others with the same ISA. For example, General Purpose GPUs are faster in processing large streams of data than a traditional CPU. Today, we already have processors with two different ISAs (e.g. AMD Fusion APUs), but only user applications use both architectures.

It would be interesting to port fast-path channels and test our design further on Barrelfish[3], a multikernel operating system capable of running on heterogeneous hardware with different ISAs, being able to utilize all computing resources that a system can have, including GPUs. Another interesting idea for a future work is to measure how much power we saved with our design, because the method is efficient only if it reduces power consumption more quickly than it reduces performance.

Bibliography

[1] National ict australia. nicta l4-embedded kernel reference manual, version n1, oct 2005. http://ertos.nicta.com.au/software/systems/kenge/pistachio/refman.pdf.

[2] Acpi component architecture. user guide and programmer reference. In *OS-Independent Subsystem, Debugger, and Utilities*, 2010.

[3] A. Baumann, P. Barhama, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. of Symp. on Oper. Sys. Principles*, 2009.

[4] J. Cong and B. Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *ISLPED'12*, 2012.

[5] P Greenhalgh. Big.little processing with arm cortextm-a15 & cortex-a7. In *White paper, ARM*, 2011.

[6] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *USENIX ATC*, 2012.

[7] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep net working - on a dependable and fast networking stack. In *DSN*, 2012.

[8] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullseni. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *roceedings of the 36th International Symposium on Microarchitecture*, 2003.

[9] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceeding SC'07*, 2007.

[10] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multicore architectures. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA10)*, 2010.

[11] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-isa cmps to save energy on operating systems. In *Journals and Magazines Micro. IEEE Volume 28, Issue 3*, 2008.

[12] D. Nellans, K. Sudan, E. Brunvand, and R. Balasubramonian. Improving server performance on multi-cores via selective off-loading of os functionality. In *Lecture Notes in Computer Science, Volume 6161/2012*, 2012.

[13] L. Niccolini, G. Iannaccone, S. Ratnasamy, J. Chandrashekar, and L. Rizzo. Building a power-proportional software router. In *USENIX ATC*, 2012.

[14] S. Panneerselvam and M. M. Swift. Chameleon: Operating system support for dynamic processors. In *Proceeding ASPLOS'12*, 2012.

[15] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. In *Journal ACM Transactions on Computer Systems (TOCS) Volume 30 Issue 2*, 2012.

[16] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack - highly efficient network processing on dedicated cores. In *Proceeding USENIXATC'10*, 2010.

[17] L. Soares and Stumm M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[18] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[19] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002.