Since the inception of the idea of an operating system managing the computer, the lives of application programmers are much easier. Operating systems provide a simpler and more machine-independent interface to the applications than the raw hardware and isolate individual applications that share the hardware. An operating system not only provides an interface to the applications it hosts, but it also provides many services which are convenient to share among applications. It isolates the applications from each other and multiplexes resources which would require exclusive access. The applications can rely on the operating system carrying out these tasks reliably with a good quality of service. Ordinary users often equate the operating systems and their user interface presented by a set of applications running on top of the operating system. Also the application programming interface (API) of the systems is often very similar. However, the internal implementation of the individual operating systems leads to significant differences between various breeds of operating systems.

For the past several decades, the internal implementation of operating systems has evolved in response to the progress in the development of hardware, the changes in the requirements by the application programmers and the growing penetration of computers in every activity of human daily life. Although the developers and maintainers of the operating systems evolved the implementation to keep up with the changing demands, certain key concepts, adopted at the time of each operating system’s inception, did not change dramatically for long and allow us to classify the operating systems based on the design decisions made long ago. These decisions were always subject to a compromise between simplicity, generality, performance and often capabilities of the hardware.

The most prevalent implementation choice became an operating system based on a monolithic kernel which runs all operating system services in the privileged processor mode, and is adopted by virtually all general purpose commodity operating systems like Linux, Windows, or the many variants of BSD, since at the time these
systems were born an operating system was a relatively small piece of software, working from time to time on behalf of the applications. The single-core chips and limited memory mainly asked for a resource multiplexor. However, the evolution and changing demands forced the systems to absorb the wider role of a provider of many crucial services, which made the original systems’ kernel grow rapidly. The monolithic approach allowed for a solution which worked and performed well, while the advances in implementation scaled to fulfill the demands of the new computerized and interconnected era. The commodity operating systems try to provide the best to all or at least the majority of applications and use cases. We use them daily on our desktops, phones, tablets or in data centers around the globe. Although there are alternatives developed for specific use cases, they are not very popular (except for embedded systems) since the commodity systems are well known, simple to use, host a huge number of applications and have massive support from hardware vendors. However, most importantly, they have performance which is not matched by other systems’ architectures and performance is the primary criterion for many customers.

At the same time, other operating systems were designed with specific interesting properties in mind, for instance reliability, security, or real-time applications. We focus on multiserver operating systems, which run all system services as unprivileged isolated processes on top of a microkernel. These systems have been designed with modularity, reliability and dependability in mind rather than focusing on performance.

Since performance is praised by many more than reliability, this kind of systems was always criticized [89] and never became a widely adopted option outside specialized areas such as embedded systems. Nevertheless, the concept of microkernels persisted and fast microkernels [70; 56; 84; 29; 31] are used in many applications, in fact, they are enjoying a renaissance because of the popularity of multicore and virtualization.

However, the performance of a microkernel is only part of the entire multiserver system performance’s problem. The mostly synchronous communication, constant switching of processes, sharing of a single processing unit and the number of processes involved in serving each application request are just some of the more important issues. The performance of the available implementations of multiserver systems was ridiculed so badly that multiserver systems are generally considered to have lost the battle. Monolithic systems rule, except in some specialized areas.

It is true that at the time multiserver systems were first created, the available hardware did not permit a well performing implementation. Therefore these systems remained niche, primarily used only in extremely demanding and constrained scenarios like controlling spacecrafts, in mission-critical applications like aircrafts, missiles or power plants. Only in such cases their superior reliability, formal verifiability, possibility of live updates without any downtime, modularity and other features, make them more valuable than highly performing commodity systems, which lack adequate design abstractions to address these concerns and, instead, resort to
scalability of implementation. This approach typically results in hard-to-maintain synchronization code to deal with the scalability challenges imposed by modern multicore architectures and the underlying cache coherence protocols. At the same time, extensive community review and testing [89] and complex error handling code are necessary.

Although building reliable systems is a daunting task, reliability must not be restricted to critical applications only. Even devices of daily use like personal computers, smart phones or tablets should really work at all times. Even though their failures rarely lead to disasters and casualties, it causes annoying user experiences like missing an important phone call or spending an overseas flight without a working entertainment system. Unexpected crashes and reboots interrupt the usage of the device and may cause loss of data and results of recent work. For instance, a crash during a long computation may not give a chance to save the intermediate results, forcing the user to start the entire computation from scratch after a reboot. In contrast to data centers, which also use commodity operating systems to deliver reliable services, personal devices cannot take easily advantage of replication and redundancy so that a fault in an instance of the operating system causes failure of the entire device. On the other hand, data centers have to overprovision not only for hardware failures, but for software failures as well. This costs money, which reliable systems can help to save. Operating systems that are reliable by design can become commodity only if they overcome their performance issues and begin to scale and perform well. In this thesis we show that it is possible, by proper design, to make a system reliable, well performing and scalable.

In this thesis we show how to modify the design to harness multicore processors which we believe change the ratio of the performance–reliability trade-off which has been in favor of the monolithic systems for many years. Multicore processors enable us to significantly reduce overheads of the multiserver design so that we can dramatically boost the performance while not compromising on reliability. In fact, we can even improve it. We use the rich expertise of the MINIX group at the Vrije Universiteit in Amsterdam acquired during many years of development of multiserver systems, namely of MINIX 3, for high reliability. We take the reliability as granted by the previous work and we enhance the design for increased and competitive scalability and performance. Christoph Lameter’s presentation at the Ottawa Linux Symposium [89] in 2007 gave ...an overview of why Linux scales and shows these hurdles microkernels would have to overcome in order to do the same. We took the extra steps to overcome the hurdles and in combination with the new multicore processors we show that multiserver systems can (i) intelligently partition the state of the services to minimize the impact of failures and (ii) scale comparably to monolithic systems. Focusing on the networking subsystem, we demonstrate that it is possible to achieve both goals without exposing the implementation to common pitfalls such as synchronization errors, poor data locality, and false sharing.
1.1 Overview of Operating System Architectures

To understand the advantages and limitations of different operating system architectures, we first present a brief overview of the two main competing architectures of operating systems, one based on a monolithic kernel and the other one using a microkernel.

1.1.1 Monolithic Kernels

A monolithic operating system is the most straightforward way to build a privileged layer between the applications and the hardware. The application can request a service from the operating system almost in the same way as making a procedure call to a library. In fact, even today the execution often continues uninterrupted in the same execution thread using the same stack to store data, and returns back to the application unless it is not possible to satisfy the request immediately. In such a case, the kernel suspends execution on behalf of the application and switches execution to another ready application. In a nonpreemptive kernel running on a single processor, there was little need for synchronization besides disabling interrupts in certain sensitive cases. The kernel was simple. Even in the case of an interrupt, the processing used to continue in the context of the interrupted application. However, as it caused high jitter, nowadays kernels postpone interrupt processing until a more convenient moment, which leads to specialized kernel threads of execution.

With the increasing availability of multiprocessors and low latency requirements, developers kept adding threads to achieve parallelism and preemption within the kernel. Many of the threads now run without direct relation to a specific process. For instance, they execute tasks periodically or asynchronously to the applications or device events. This significantly increased the complexity of the kernel. On one hand, various threads of execution could run in parallel (virtually or actually), on the other hand developers had to deal with safeguarding accesses to unprotected shared data, which created the problem of retrofitting the required synchronization into the existing code bases of the quickly growing operating systems. Simple solutions—for example the big kernel lock of Linux—worked, but did not scale. However, replacing it with finer grained and scalable solutions proved to be an extremely complex and error prone task. Although the advances in implementation of the synchronization, annotation of read-mostly or CPU local data, etc. allowed monolithic systems to scale, the shared-everything model requires the developers to be extremely cautious and poses deep knowledge of subtle details of the systems to produce deadlock free code.

Monolithic systems contain drivers—code plugged into the kernel to control peripheral devices. The plethora of device types and different vendors requires specific code for many of the devices. These drivers are usually supplied directly by the manufacturers or enthusiasts who just need their specific model to work. Although the core developers of the open source systems try hard to refactor, unify and reuse lots of code, the developers of the drivers are not bound by anything but
the ever-changing internal APIs. Therefore everyone is free to introduce their own new threads to carry out tasks so far not anticipated by the unified code, as well as new locks to protect data structures not previously shared. In fact, the drivers can do virtually anything. Even though the drivers are often less safe than the rest of the system’s kernel [41], they are not isolated from the rest of the kernel whatsoever and any of their failures easily propagates to the rest of the system causing a fatal crash.

One of the most widely used monolithic kernels is Linux. In contrast to proprietary systems, which are distributed without their source code, development of Linux is open to the world with a vast community of pure enthusiasts as well as employees of large companies. Due to its open source nature, everybody can study the system and its use is hugely popular due to its royalty-free license. Therefore we can find Linux in practically every domain of computing, ranging from home routers, smart phones and PCs to servers, data centers and super computers. Although there are other open source operating systems such as FreeBSD, none match Linux in popularity and deployment. This makes Linux the de facto standard system for comparison.

At the same time, evolution and implementation of Linux and other open source monolithic systems is driven by conflicting needs of different users. What suits one does not need to suit others and therefore the implementation is necessarily a compromise.

While the amount of human effort, the rich set of features and, of course, the money invested, makes it hard to compare apples with apples in the case of research in operating systems. We nevertheless use Linux to show that performance of our research system is in the same league as the state-of-the-art commodity systems.

1.1.2 Microkernels

In contrast to monolithic kernels, a microkernel contains only the essential mechanisms that isolate processes, switch between their execution, manage low-level resources and grant access to peripherals, while user-space processes implement all policies. Unlike in a monolithic system, applications do not request the system service by switching to the privileged mode where it continues with execution of the system code. Instead, the applications must ask other user-space programs to provide the service. All processes communicate with others by passing messages, which the kernel dispatches, while it suspends the sending process until a reply with results is ready.

Using a microkernel, the systems can have several forms. For instance, L4Linux uses a single system server, in fact a paravirtualized Linux kernel, for implementing all of the operating system services. All applications, even though running natively on top of the L4 microkernel, pass their system calls to the system server, much like if they were running on regular Linux. While the single system process has many of the disadvantages of the original Linux, the key difference is that the system server can use native L4 drivers, processes isolated from the system server, to access
CHAPTER 1. INTRODUCTION

devices. Therefore a fault in any driver, possibly provided by an untrusted third party, does not immediately cause a fault in the main system server as well.

A big step towards even more reliable operating systems has been achieved by the seL4 [84] project. It is the first formally verified microkernel. In other words, the microkernel has been proven to work according to the specification. Although it does not guarantee on its own that the system running on top of this kernel is reliable, dependable or safe, it creates a solid point for building such systems, while monolithic systems cannot rely on anything of this kind.

A microkernel can host multiple independent instances of the system server like L4Linux, de facto serving as a hypervisor. In fact hypervisors like Xen [29] or Nova [142] are, in a sense, custom built microkernels and Xen’s Dom0 serves a similar purpose as the native drivers of L4Linux. However, an entire Linux is used to provide the drivers and rich management. Unfortunately, this carries the same isolation and reliability problems as a monolithic system. Therefore a disaggregation of the Dom0 has been proposed [108].

Moreover, VirtuOS [112] demonstrated recently that it is possible to build a single system image out of several monolithic kernels that run on top of the Xen hypervisor instead of a microkernel. In this way, the virtualization allows a semi-decomposition of a monolithic system. Even though each virtual machine (domain) runs an entire Linux kernel, it is responsible for a certain system task only. Although this design introduces communication overheads between the domains, it preserves the performance of each isolated subsystem, while it prohibits propagation of faults between the domains. However, the current implementation includes only I/O domains for networking and storage and the primary domain still uses a massive portion of the huge kernel. In a way, VirtuOS improves reliability of a monolithic system using monolithic kernels as building blocks to mimic the microkernel approach.

In contrast, microkernels allow for a truly clean slate design of the entire system, where individual services run as a collection of isolated processes, the multiserver systems. Although the design is clear, reliable, modular and versatile—Linus Torvalds himself stated in the famous Torvalds–Tanenbaum debate that... the design is superior from a theoretical and aesthetical point of view 1—it contains performance issues that were considered inherent. The fact that handling of each application request or external event can involve multiple processes, each running for a brief moment only, causes high overheads for managing execution of the processes. Not only does the microkernel need to process many messages, schedule the processes and keep saving and restoring their contexts, but the hardware itself is underutilized by the frequently switching streams of instructions which flush the CPU internal structures as each process needs different data in caches, newly populated TLB and reestimated branch predictors. For these reasons, the single core processors of the past were a far better match for monolithic systems when it came to performance. Although the switching between applications and the privileged kernel has similar

---

1Re: “Linux is obsolete”, January 29, 1992
1.2. NEW MULTICORE HARDWARE

Hardware issues, the switching is orders of magnitude less frequent and hence the performance hit is smaller. Nevertheless, recent studies show \([138; 68]\) that even in the monolithic systems the penalty is significant.

In fact, in the past, to avoid the cost of switching between the monolithic kernel and user space, some web servers \([94]\) were placed directly in the Linux kernel to avoid the cost of mode switching. These days, developers of D-Bus \([5]\) are pondering moving the entire interprocess communication mechanism into the kernel \((kdbus [8])\). Similarly, to minimize the “inherent” overheads, developers of microkernel based systems take shortcuts and diverge from the clean model to a hybrid design by placing more than the necessary minimum into the kernel \((for example the XNU kernel of Mac OS X)\), while only retaining the option to run the device drivers in user space, as they are often the biggest source of bugs, possibly compromising the rest of the operating system.

Not only was the hardware initially more suitable for building monolithic systems than other system architectures, the hardware evolution was also partly driven by the most widely used type of systems and software in general. Specifically, the hardware vendors implemented features to improve the performance of the monolithic systems. For instance, the SYSCALL (AMD) and SYSENTER (Intel) instructions make the transfer between execution in user space and in the kernel faster. However, the hardware did not improve much the support for messaging.

On the other hand, we must highlight the good properties of the microkernel based operating systems, namely their reliability and dependability. Jonathan Shapiro stated in 2006 in a Linus Torvalds rebuttal \([135]\) that *ultimately, there are two compelling reasons to consider microkernels in high-robustness or high-security environments: (i) There are several examples of microkernel-based systems that have succeeded in these applications because of the system structuring that microkernel-based designs demand. (ii) There are zero examples of high-robustness or high-security monolithic systems. Multiserver systems do not only contain faults in isolated components. It is also possible to update different parts of the system while it is running without any downtime, allowing live updates before a known bug causes a fault or before someone exploits a programming error. Although it is possible to patch a running Linux kernel \([28]\), these patches are largely limited to critical security issues and the scope of the fixes is narrow as it is complicated to bring the share-everything running kernel into a quiescent state. In contrast, due to the isolation of the multiserver system components, it is possible to “pause” a system server, transfer its state to its new incarnation, unplug the old one and carry on with the new version. It is our goal to make these properties part of commodity operating systems too.*

1.2 New Multicore Hardware

Hardware has changed dramatically in the past decade. Instead of a single execution unit with ever increasing clock speed, the frequencies of the modern chips have
leveled off. Instead, they have multiple cores on a single die and each core can have multiple execution threads. For instance the SPARC T5 (announced in 2012) features 16 cores. Each core has 8 threads out of which 2 can execute simultaneously. The recently\(^2\) unveiled SPARCs double the number of cores. This makes for a total of unprecedented 256 hardware threads per chip! Likewise, the newly announced Intel Xeon Haswell-based chips have up to 18 cores with 2-way hyperthreading, and the Xeon Phi coprocessor offers up to 61 cores, while the next generation Knights Landing is supposed to have up to 72 cores with 4 threads each, available in a coprocessor as well as a socket version. This is an enormous increase in parallelism that is hard to tame outside the world of supercomputing and data centers.

The question we investigate in this thesis is how much the new multicore processors with their available parallelism change the game in the arena of operating systems in favor of the multiserver systems. In contrast to the old, single core processors, having more parallel execution units on the same chip lets the system services run side by side without the need for switching between them. We investigate the idea that having a vast number of hardware threads, each performance critical system process of a multiserver system can possibly use its own core or a thread. This way, the processes can execute anytime they need. The scheduler does not need to guess which of the processes is the best to run next and which one to preempt to run another one.

A multiserver system resembles a true distributed system. However, the advantage of running the system within a single machine on a single chip is reliable and trustworthy communication. The interprocess messages cannot get lost or spoofed in the Internet. On the other hand, a multiserver system cannot take full advantage of the new hardware out of the box. The cost of cross-core kernel-based interprocess communication is higher than on a single core chip and thus it still remains a significant bottleneck. On a single core chip, the processes can exploit the causality of synchronous messaging. In case the receiver is able to receive, the microkernel can use the current location of the message, for example the machine registers, and pass it to the receiver in-place and schedule it to run immediately, as done by the family of L4 microkernels. In contrast, synchronous cross-core messaging requires cross-core notification, since the microkernel running on the “receiving” core might be sleeping or executing an application rather than actively monitoring whether it should receive a message from another random core. The available notification mechanisms are interprocessors interrupts (IPI), which the sending core needs to set up in an interrupt controller, while the receiving core must handle it as a generic interruption. Therefore the receiver is not only returning from the privileged mode to user space, but the core must wake up first and handle the exception. This is costly.

In this work we show that it is possible to radically reduce the cost by excluding the kernel from the fast path\(^3\) and moving the communication entirely into user space. However, we can do that efficiently only if the communicating processes run

---

\(^2\)As of the time of writing in Fall 2014

\(^3\)Path that handles the most commonly occurring tasks more efficiently than the ‘normal’ path.
in different hardware contexts. This allows the system to stream messages throughout the system. We demonstrate that with some help from the hardware, we can make the user-space communication efficient and we also stress that there is more the designers of future hardware can do to help us to further improve the communication.

1.3 Network Stack

To demonstrate the viability of our idea we focus on the network stack, a subsystem that connects the operating system and the applications it hosts to the outside world. The network stack is a performance-critical component in modern operating systems. At the same time, it has a long history of reliability issues, and is widely recognized as a complex subsystem. Therefore enhancing reliability of networking without compromising its performance is important.

Development of monolithic systems concentrates primarily on improving performance to keep up with the speed of the interconnection technologies. Handling gigabit rates was challenging years ago, 10 gigabit speeds are the norm nowadays, while the industry is moving to 40 and 100 gigabit links. Multiserver systems are able to deliver the reliability, but unfortunately the price is very high and handling gigabit traffic at wirespeed typically remains a wishful goal.

The key limitation of multiserver systems in achieving high bitrates is synchronous communication between the network stack and the drivers and data copying from the network stack to the drivers and vice versa. While the CPU is busy, the network interface does not receive enough packets to keep the link fully utilized since per-packet overheads in software create large interpacket gaps. Even though microkernels often offer some sort of asynchronous communication by means of notifications (a mechanism similar to hardware interrupts), the sending process must interrupt its stream of execution and ask the microkernel to deliver the notification, after which the receiving process gets likely scheduled. This effectively synchronizes the processing. In a way, the synchronous communication is not much different from the procedure calls within a monolithic system, but the overhead of such calls is orders of magnitude cheaper. We show that the multiserver system can counter this problem by using asynchronous user-space communication between components on different cores, which allows pipelining of processing without excessive gaps and bursts. The user-space communication depends on polling or, preferably, hardware support for low overhead cross-core notification which does not disrupt the execution streams.

The monolithic system architecture not only has less internal overhead, it also “naturally” scales to multiple cores by making the code multithreaded. The one single-threaded component per function paradigm in a multiserver system does not enable such a straightforward approach. Although in the past, having one single-threaded process to handle all network traffic did not seem prohibitive, it is no competition to the multithreaded network stacks of commodity operating system running
on multicore servers in data centers. A single-threaded component simply cannot use the power of multiple cores. Indeed, it is possible to make the single networking component multithreaded as well, however, such a component becomes complex, deadlock and error prone. At the same time, the scalability issues of monolithic systems [38], due to contention on locks and problematic data sharing, make multithreading not so “natural” as it may seem at first glance. In contrast, we show that it is possible to scale the network stack of a multiserver system by running several fully isolated replicas of the network stack. Although this solution requires some assistance from the network cards, contemporary hardware has almost all the necessary features and the remaining may well emerge in the near future as also other groups express their interest in them. They advocate for the operating system to become a control plane [120; 33], while the hardware takes the responsibility for multiplexing and isolation.

The way we see the novel reliable and scalable operating system design fundamentally differs in how the CPU cycles are used. In a monolithic system, each CPU core is used for general purpose, hosting both the applications and the operating system’s kernel. Depending on the application and its requests, the CPU executes the required part of the kernel. This means that the core’s cycles are split between the system and the applications and all of the cycles can be used meaningfully. Similarly, applications share the cores with the operating system servers. Although some functions may execute on a different core than where the application is running, another application or a system service can use that core in the meantime. It looks obvious that in a system which dedicates some of the cores to some of its components (as we propose), it can happen that many cycles cannot be used as no other process can run on the dedicated core even when it is not fully utilized. Moreover, the cores dedicated to the system cannot be used by the applications. However, at a second glance, the reality is different. We will show that the usage of CPU cycles is just distributed differently.

Specifically, in a traditional system, applications can use only a fraction of a core’s cycles. For instance, in network-intensive workloads, the network processing in the kernel can use more than 70% of each core while the applications can use only the remainder, effectively using only less than 30% of all the available cycles [81] in the machine. The application processing as well as the processing in the kernel is distributed across many cores. In contrast, in the case of dedicated cores, the application processing as well as the processing in the system is consolidated on the system cores and the application cores respectively. If the application can use 30% of all the cores, it has the same number of cycles at its disposal. We show that when the load is high, the sum of cycles available for the applications is the same as in the case of a more traditional system and the processing is smoother, even though the distribution of the usage looks a little counterintuitive.

Indeed, a problem may arise in the case of mixed workloads when the cores dedicated to the network stack are not fully utilized. The answer to this problem is two-fold. Cores with multiple hardware threads allow us to coalesce different com-
ponents on different threads of the same cores. While a traditional multiserver sys-

tem requires the microkernel and the scheduler to switch between the processes on a

core, multithreaded cores interleave the execution streams with much finer granular-

ity without additional involvement of software, achieving the illusion of processes

running simultaneously but with much smaller overhead. The hardware scheduler

switches the streams when they have data available, which makes the use of the hard-

ware more efficient as cycles are not wasted when the execution units stall waiting

for the fetches from main memory. This setup may result in some interference in

cache. However, if this is limiting, the load is too high and we have to stop coales-

cing. An orthogonal option is to use heterogeneous processors that can offer many

simpler cores than general purpose processors. The system can use the higher num-

ber of cores for finer grained resource allocation and for better power management

as the “smaller” simpler cores tend to be more power efficient. We show that the

performance of such cores is sufficient for running even highly loaded system tasks.

1.4 NEWTOS

To prove the concept and to evaluate the ideas we implemented NEWTOS, a system

based on the MINIX 3 multiserver system. We call the system NEWTOS because

like newts, little amphibians that have the unique ability to re-generate limbs and

organs in case of injury4, NEWTOS can survive crashes and recover parts of the

system.

We extended MINIX 3 by adding support for multicore chips and we have reim-

plemented the user-space network stack. Instead of the original single process, we

separated processing of individual networking protocols into isolated processes. The

split along the protocol boundaries is a natural choice. This increases pipelining and

separates functions that are easier to recover from failures and the ones with a large

state which is hard to preserve. we introduced asynchronous user-space commu-

nication using the so-called communication channels (essentially message queues)

between these components that bypass the microkernel. We also abandoned the

mostly synchronous communication between the network stack and the network

device drivers and replaced it with the same user-space communication. In addi-

tion, we extended the communication all the way to the applications by redesigning

the network sockets. Specifically, We expose the socket buffers to the applications.

Thus, applications can check whether the buffer is full or empty and access the data

directly. This allows for uninterrupted execution of the applications by avoiding

almost all of the legacy trap-based system calls which were so far considered the

main bottleneck. To make the communication channels work efficiently, the ker-

nel provides the system processes with a new way of blocking (using the MWAIT

instruction). However, on some architectures, the user-space processes can even do

it on their own without any assistance from the microkernel. To make the network

stack scalable, NEWTOS can run multiple copies of the entire stack. This fact is completely transparent to the applications while it allows spreading the load among multiple cores. The number of the network stack instances is flexible and adjustable to the current workload.

1.5 Contributions

Our contributions are several fold:

1. We show that the often touted performance issues of multiserver systems are not inherent. Specifically, we show that with modern multicore processors many of the old performance bottlenecks of multiserver systems can be removed to make them competitive in performance (but better in reliability) in comparison with commodity monolithic operating systems.

2. We implemented a novel network stack for a multiserver system which improved the network throughput from a little more than 100 Mbps to 10 Gbps. This throughput is only limited by our testing hardware and not by the structure of the network stack.

3. We also introduced novel communication between the applications and the network stack, which can avoid almost all network related system calls an application that uses the network must execute in other operating systems. Under high load we can reduce the number of system calls by more than 99%. In other words, almost all the communication occurs in user space and bypasses the microkernel.

4. We show that isolation and partitioning have better scalability and reliability properties than the shared-all model of monolithic systems. We can run independent isolated instances of the network stack to scale the throughput of the system as the load demands, while the instances do not share any data structures, therefore there is no contention in any synchronization.

5. We show that multiserver systems can adapt to the future heterogeneous multicore architectures as we can run their parts on the most appropriate types of cores and the best core is not necessarily the fastest one in the system. On the contrary, we show that slow cores can deliver unexpected performance.

6. We demonstrate that operating systems can have both reliability and performance without trading one for the other. The more performance we gain, the more we can afford to further decompose system servers and their state which makes recovery from faults simpler. In addition, running multiple instances of the same system servers means that any fault does not bring down the entire service they provide.
1.6 Organization of the Dissertation

The work has been published and submitted to refereed conferences and workshops and the remainder of this dissertation, which includes this work, is organized as follows:

Chapter 2 presents a new research system NEWTOS, which we use for prototyping our ideas. It is based on MINIX 3, however, it diverges strongly in the way it uses the available cores and how the system servers communicate. NEWTOS shows that it is advantageous to pin the performance sensitive servers and drivers to their own cores, which allows them (i) to execute without interleaving with other processes, (ii) communicate asynchronously without involvement of the microkernel and (iii) pipeline their work, hence increasing parallelism and, in turn, (iv) improve reliability by allowing the system to split the services into even more fine grained and isolated components. NEWTOS demonstrates that multiserver systems and multicore processors are a great match. To support our claims we cover in detail the implementation and evaluation of its network stack, since it is a subsystem, which must process large number of messages generated by intensive workloads.

Chapter 2 appeared in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*

Chapter 3 discusses how NEWTOS can achieve the best performance on heterogeneous processors, which allows each server to run on the most suitable type of core. In addition, it shows that the fastest and the “most powerful” cores are not necessarily the most suitable ones and that slow (or *wimpy*) cores can deliver superior performance for the system processes. The heterogeneous multicores have the potential to offer more cores on a die of the same size than their homogeneous counterparts in the future, however we use commodity chips in combination with frequency scaling to slow down some of the cores to emulate wimpy cores of heterogeneous platforms and we show the potential for energy savings and improved power/performance ratio.

Chapter 3 appeared in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC’13)*

Chapter 4 focuses on efficient resource usage of multicore, heterogeneous and over-provisioned processors. Most of the prior work focused only on scheduling applications in such environments. In contrast, we show the operating system must get the same attention. The multiserver systems can embrace such processors more easily than other systems, however, it is not possible without changing the way the operating system itself perceives its system processes since they are different than applications. The scheduler must not only passively observe metrics provided by the hardware and guess what is the best for each of the system servers. They must not be opaque processes anymore.
The scheduler must know each of the server’s role and requirements. At the same time, the servers must supply the scheduler with performance indicators and the scheduler itself must take an active part in evaluating the data and re-configuring the system depending on the current load, available resources and power budget. We propose “profile based scheduling” as the simplest method to match current readings of the indicators and good predetermined configurations.

Chapter 4 appeared at *The 4th Workshop on Systems for Future Multicore Architectures (SFMA 2014)*

**Chapter 5** explores the possibilities and benefits of removing a significant overhead of operating systems—the system calls—by running parts of the operating system on different cores than the applications. Prior work shows that system calls disturb execution of applications by interleaving them with the execution of the operating system, which invalidates internal structures of the processor. NEWTOS as well as modifications of some commodity systems, allows parallel execution of applications and the operating system on disjoint cores. NEWTOS demonstrates that it is possible to extend the fast user-space inter-process communication and signaling, which the operating system uses among its servers, to the applications. We focus on the BSD sockets API system calls and we show that exposed socket buffers based on producer–consumer queues can avoid the vast majority of the system calls as the applications can resolve them immediately on their own and only use true system calls when they need to block. In contrast to other approaches to minimize the socket system calls, we entirely exclude the kernel from the fast path, while we preserve backwards API compatibility.

Chapter 5 appeared in *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS ’14)*

**Chapter 6** discusses the limitations of commodity systems for the increasingly high scalability and reliability demands of modern applications. We show that their architectures lack adequate design abstractions to address these concerns and instead resort to scalability and reliability of implementation. We demonstrate that a simple and a principled OS design driven by two key abstractions—*isolation* and *partitioning*—is effective in building scalable and reliable systems while solely relying on scalability and reliability of design. To substantiate our claims, we apply our ideas to the NEWTOS’s network stack and present a clean slate implementation called NEAT, and discuss the architectural support necessary to deploy our solution in real-world settings. We show that our design can (i) intelligently partition the network stack state to minimize the impact of failures and (ii) scale comparably to Linux, but without exposing the implementation to common pitfalls such as synchronization errors, poor data locality, and false sharing.

Chapter 6 is *under submission*
Chapter 7 concludes the dissertation by summarizing the results and discussing opportunities for future research directions.