Hardware has changed dramatically in the past decade and made void the assumptions that were true when most of the current commodity operating systems were conceived. At the same time, the demands on the operating systems have increased as computers have penetrated into our daily lives. Computers should not require an ordinary user to be an experienced operator anymore. At the same time, they should be reliable because the large organizations manage such a vast number of computers that frequent failures demand a lot of well paid staff. Although reliability of commodity operating systems has increased over the years (for example, the infamous Windows “blue screen of death” is not as common a sight anymore), developers paid much more attention to improving performance, even though crashing machines can also have severe overall performance and financial implications. In fact, reliability has been specifically addressed only in niche domains, where compromising on reliability is impossible due to fatal consequences of any crash and the need to pay the high price for the extreme reliability is indisputable.

In this thesis, we explored whether it is possible to take advantage of current hardware trends and modify the reliable multiserver operating system design in such a way that it can become an option for building commodity operating systems, which can have both performance and reliability and to achieve it without making compromises on either side. Such a system can improve the experience of average computer users as well as reduce the cost of overprovisioning in data centers or reduce the runtime of long running complex scientific computations. We demonstrated that it is possible at least for the network stack!

We achieve our goal by exploiting the fact that current multicore processors can execute different parts of a multiserver operating system simultaneously without the need of switching between them, thus removing the biggest performance overhead of multiserver systems without changing their general structure. However, this approach is not a straightforward solution that works out of the box and our work discusses the logical and practical details of a reliable, dependable, well performing
and resource-aware operating system, which we call NEWTOS.

First, we demonstrated that the communication between servers and drivers of the system is prohibitively expensive and blindly using multicore processors to run the servers on dedicated cores does not make the system perform better. To the contrary, the cost of cross-core communication provided by a microkernel increases. To overcome the problem, we introduced pure user-space communication between pairs of cooperating processes. The microkernel is excluded from passing messages and only serves as a provider of reliable communication which allows the processes to setup the communication channels in a trustworthy way. In addition to avoiding the microkernel overhead of message passing, the new user-space communication channels are asynchronous, so the communicating processes do not need to wait for a rendezvous and can proceed with processing independent requests without blocking.

To demonstrate the benefits of the new communication mechanism and the distribution of system processes across the available cores, we reimplemented the network stack, a critical part of modern operating systems for performance and reliability. Even though we could have used another subsystem, we opted for the network stack because currently well established multigigabit links can generate load that is beyond what has ever been thought that a multiserver system could handle. In contrast, a subsystem like the storage stack is limited by the speed of the storage devices and only emerging flash-based technologies can sustain similar data transfer rates. In addition, due to the growing popularity of network attached storage (NAS), the network stack is becoming a vital part of the storage stack itself.

We have measured that slow and, above all synchronous, messaging limits the speed of networking to a few hundreds of megabits per second only. This is unbearably slow even for home users of NAS, for example for processing photos, videos and backups. The novel network stack of NEWTOS has dramatically raised the bar and it shows that a multiserver system, much like other commodity operating systems, is able to process data at multigigabit rates.

Moreover, while we have increased performance, we simultaneously increased reliability of the entire operating system. Using the same principle of breaking up a monolithic system into a collection of isolated servers, we decomposed the monolithic network stack into several components along the boundaries of the individual protocol layers. This simplifies crash recovery since some components have little or no state making them easy to restore. At the same time, we can use specific methods with varying performance overheads to protect components with large and frequently changing state. The key benefit of this decomposition is that a crash in an easily restorable part of the network stack does not directly affect the more complicated components. We tested the design by injecting faults while processing a gigabit TCP stream. Crashing a packet filter has negligible effect on the stream performance as it recovers instantly. However, recovering from an IP crash takes slightly more than a second. Although the IP component recovers almost instantly, the network interfaces, which are not manufactured with software restartability in mind, need a reset to flush memory descriptors provided by IP. Unfortunately, reset-
ting the hardware takes a lot of time.

Running some system processes on dedicated cores naturally limits the number of cores available to other system processes and, above all, to the applications. This contrasts with the common understanding of how operating systems work based on the prevalent experience of most users with monolithic systems. Although our assumption is that the number of cores of future processors will be less limiting, we investigate the benefits of running a multiserver system on cores that feature more hardware threads as compared to cores, adding threads is easier, cheaper and the system processes often need only a hardware container for their context. At the same time, we also emulate heterogeneous architectures to demonstrate that smaller and simpler cores, which we call wimpy cores, are sufficient for running system processes even when the system is relatively highly loaded. In fact, we discovered that it is often beneficial to run some parts of the network stack on slower cores to achieve higher throughput. This counterintuitive result is especially true for the network device drivers.

New products of major chip manufacturers as well as recent research publications indicate that we may see more heterogeneous architectures in the future. We show that a multiserver system is well suited for such architectures as it is easy to run different servers of the operating system on the most appropriate cores. However, this requires the scheduler to carefully assess optimal placement (in respect to the workload, available resources and energy budget) of the system processes. Since even the scheduler is an isolated process, which can run any time it needs to, it can gather rich statistics from different parts of the system and run nontrivial evaluation algorithms to make good scheduling decision.

Another limitation of running parts of the network stack on dedicated cores is the fact that a single component cannot handle more work than what is possible to handle on a single core. Even though our network stack employs multiple cores, and thus can handle more than a single component network stack could have, the scalability is limited by the most demanding component. At the same time, we want to avoid multithreading. To scale the network stack, we devised a method of running multiple copies of the network stack. Each copy or replica works on its own as the copies are all strictly isolated and do not communicate with each other. This avoids complicated synchronization of shared data structures, which requires elaborate (and hence error prone) synchronization schemes and bouncing of data between caches of individual cores. It is exactly this issue that complicates scalability of monolithic systems that are inherently multithreaded.

Running multiple independent replicas of the network stack allows the multiserver system to scale and naturally increase its reliability as a failure in one of the replicas does not render the entire network stack unusable. This is especially important for TCP processing since the TCP components have large, complicated and frequently changing state. A crash in one of the TCP replicas means that only a portion of the state is lost or requires recovery. At the same time, applications can use the other replicas to serve their needs. The downside of running multiple replicas is
that the system must manage the number of available replicas and match it with the desired load, reliability and available resources. This can be configured manually, similar to how administrators manually tune the Linux network stack for high performance of given workload on the given types of machines. However, our goal is to make the system self-adjusting. To be able to multiplex efficiently the data streams among the running replicas of the network stack, we need support from the network cards. With the proliferation of virtualization comes the need to spread the work among many cores in commodity operating systems and manufacturers enhanced the network cards with many clever features that enable them to take on the isolating role of the operating system. We use these features and we argue that some tweaking is necessary for these features to fully support building of reliable systems.

Last but not least, we demonstrated that the user-space messaging we use within the network stack to communicate between its parts is further extendable as we use it to improve communication between the operating system and the applications. Applications can use this extended mechanism to request operating system services without expensive system calls that disrupt their execution. We use the model of the user-space communication channels to build network sockets that expose information to the application that was previously provided only by the operating system upon request. Using the new sockets, the applications can decide themselves which socket is ready. That means, when a socket has data available or can accept new data for sending. Testing the state of the socket has negligible overhead in comparison to legacy nonblocking system calls as we can avoid more than 99% of the system calls when the system is highly loaded. Therefore the application can speculate more aggressively as the penalty for a failure is low. We have presented that legacy event-driven servers can take advantage of these sockets without any code modifications and achieve better performance running in NewTOS than in Linux.

We successfully demonstrated that it is possible to build systems that perform “on par” with commodity operating systems, while offering a huge competitive advantage in their reliability and dependability. This combination makes them unique and should make them widely acceptable in many areas of computing.

Future Work

Although NewTOS shows the direction to fast and reliable computing, plenty of work remains to be done in the future.

Beyond the network stack Although our proof of concept shows that our principles can improve the network stack of a multiserver system, wider acceptance and deployment require us to apply the same principles to other parts of the operating system as well. For example, the storage stack in multiserver systems is already composed of several components (virtual file system, the actual file systems and storage device drivers), while further decomposition for the sake of modularity [27] and reliability is desirable. As flash-based storage is
gaining momentum, it will be able to stress the operating system in the same way as high speed networks do. In fact, PCIe flash cards can do that already.

**Self-adjusting system** As we spread the multiserver system across many cores, it is necessary to develop smart algorithms for intelligent and efficient management of available resources. Our current work only provides the required mechanisms to implement such schedulers and monitors that can adjust the whole operating system to the changing runtime conditions. We must take the user’s assistance out of the loop and automate the process of evaluating the system online. Such schedulers must learn from the current usage patterns and performance statistics to devise schemes for the best placement of system components on the available CPU cores depending on the mix of applications, workloads, energy budget constraints and user preferences. Due to the similarity of multiserver systems to distributed systems, we can base our future research on the methods applied in distributed computing, although at a much finer level. We would like to apply machine learning and data mining techniques to analyze the rich set of values that we can collect from various system servers.

**Hardware support** We pointed out many times in our work that we can marry performance and reliability only thanks to the advances in microprocessor architectures. Nevertheless, we also stress that the hardware can do significantly more to help software to carry out its tasks more efficiently. In the first place, having more support for message passing is a pressing matter. For instance, we have used the **MWAIT** instruction to implement efficient and power-aware cross-core communication. However, this instruction is specific to the x86 architecture. To the best of our knowledge, other architectures lack similar mechanisms. At the same time, it is not possible to use this instruction in user space on the Intel chips, while it is possible on some AMD chips. To make our system easily portable across these two platforms, we resorted to using a kernel call to execute the instruction. Removing this overhead will likely lead to a reduction in latency of our system under low load.

In addition, we would like to explore a finer grained control over the privileges of user-space processes. It is possible to set many features on virtual machines. However, in the case of privileged versus unprivileged code, it is either all or nothing. The system should be able to grant some of its servers access to certain CPU features that ordinary applications do not need, but which do not conflict with isolation in constrained conditions. For instance, a system server on a dedicated core should not be restricted from pausing it.

**Efficient polling** Even if we suspend a core which has no work to do in user space, waking up the core again takes some time. Although we cannot know whether the next request arrives almost immediately or whether there will be an extended period of idle time, the system components could adapt its polling intervals based on the recent stream of requests. The server could keep polling
a little longer if it estimates interarrival gaps shorter than the time needed to suspend and wake up the core or when it does not save energy. This of course differs for each platform.

**Portability** Due to the MINIX 3 legacy, NEWTOS runs only in 32-bit mode of x86 processors. Although we can use the different features Intel and AMD provide, for example the number of threads per core, there are other interesting architectures with different types of cores, number of threads, types of pipelines and unusual instructions. Therefore portability across various platforms is extremely important for further research.

**Shared memory and cache coherency** Our current implementation of the fast user-space message passing relies on shared memory and using shared memory stresses the cache coherence protocols as data are being produced on different cores than where they are being consumed. We believe that a finer grained control of caching could result in (i) faster communication and (ii) less interference with other data in the cache. In fact, we only use cache coherent memory out of convenience as our current evaluation platform offers it. In an extreme case, we may lack the cache coherency altogether. The explicit message based communication is well suited for such environments as we know exactly when we want to make our memory modifications globally visible and the remote side does not need to see partial changes at all. At the same time, the producer does not need to cache the messages since they are not read again.

**Application to other operating systems** The most important part of our future work is to demonstrate that our design principles are a viable option on the production level. Since the engineering effort of writing a production grade operating system from scratch is enormous, we see the more likely direction in applying parts of our work to the existing commodity operating systems. For example, running our network stack side by side with the Linux kernel, only substituting this particular functionality. Although the entire system would not become reliable at once, enhancing reliability of a key subsystem is already a big leap forwards, even though any fault in other parts of the kernel would still bring the entire system to a halt.