Keep Net Working - On a Dependable and Fast Networking Stack

Tomas Hruby Dirk Vogt Herbert Bos Andrew S. Tanenbaum
The Network Institute, VU University Amsterdam
{*thruby,dvogt,herbertb,ast*}*@few.vu.nl*

***Abstract*—For many years, multiserver[1] operating systems have been demonstrating, by their design, high dependability and reliability. However, the design has inherent performance implications which were not easy to overcome. Until now the context switching and kernel involvement in the message passing was the performance bottleneck for such systems to get broader acceptance beyond niche domains. In contrast to other areas of software development where fitting the software to the parallelism is difficult, the new multicore hardware is a great match for the multiserver systems. We can run individual servers on different cores. This opens more room for further decomposition of the existing servers and thus improving dependability and live-updatability. We discuss in general the implications for the multiserver systems design and cover in detail the implementation and evaluation of a more dependable networking stack. We split the single stack into multiple servers which run on dedicated cores and communicate without kernel involvement. We think that the performance problems that have dogged multiserver operating systems since their inception should be reconsidered: it is possible to make multiserver systems fast on multicores.**

***Keywords*-Operating systems; Reliability; Computer network reliability; System performance**

I. Introduction

Reliability has historically been at odds with speed—as witnessed by several decades of criticism against multiserver operating systems ("great for reliability, but too slow for practical use"). In this paper, we show that new multicore hardware and a new OS design may change this.

Reliability is crucial in many application domains, such as hospitals, emergency switchboards, mission critical software, traffic signalling, and industrial control systems. Where crashes in user PCs or consumer electronics typically mean inconvenience (losing work, say, or the inability to play your favorite game), the consequences of industrial control systems falling over go beyond the loss of documents, or the high-score on Angry Birds. Reliability in such systems is taken very seriously.

By radically redesigning the OS, we obtain both the fault isolation properties of multiserver systems, and competitive performance. We present new principles of designing multiserver systems and demonstrate their practicality in a new network stack to show that our design is able to handle very high request rates.

The network stack is particularly demanding, because it is highly complex, performance critical, and host to several catastrophic bugs, both in the past [14] and the present [4]. Mission-critical systems like industrial control systems often cannot be taken offline to patch a bug in the software stack—such as the recent vulnerability in the Windows TCP/IP stack that generated a storm of publicity [4]. When uptime is critical, we need to be able to patch even core components like the network stack *on the fly*. Likewise, when part of the stack crashes, we should strive toward recovery with minimal disturbance—ideally without losing connections or data.

In this paper, we focus on the network stack because it is complex and performance critical, but we believe that the changes we propose apply to other parts of the operating system as well. Also, while we redesign the OS internals, we do not change the look and feel of traditional operating systems at all. Instead, we adhere to the tried and tested POSIX interface.

Contributions: In this paper, we present a reliable and well-performing multiserver system NewtOS[2] where the entire networking stack is split up and spread across cores to yield high performance, fault isolation and live updatability of most of the stack's layers. We have modified Minix 3 [1] and our work has been inspired by a variety of prior art, such as Barrelfish [5], fos [43], FlexSC [39], FBufs [12], IsoStack [37], Sawmill Linux [16] and QNX [33]. However, our design takes up an *extreme* point in the design space, and splits up even subsystems (like the network stack) that run as monolithic blobs on all these systems, into multiple components.

The OS components in our design run on dedicated cores and communicate through asynchronous high-speed channels, typically without kernel involvement. By dedicating cores and removing the kernel from the fast path, we ensure caches are warm and eliminate context switching overhead. Fast, asynchronous communication decouples processes on separate cores, allowing them to run at maximum speed.

Moreover, we achieve this performance in spite of an extreme multiserver architecture. By chopping up the networking stack into many more components than in any other system we know, for better fault isolation, we introduce even more interprocess communication (IPC) between the OS

[1]Operating systems implemented as a collection of userspace processes (servers) running on top of a microkernel

[2]A newt is a salamander that, when injured, has the unique ability to re-generate its limbs, eyes, spinal cord, intestines, jaws and even its heart.

components. As IPC overhead is already the single most important performance bottleneck on multiserver systems [26], adding even more components would lead to unacceptable slowdowns in existing OS designs. We show that a careful redesign of the communication infrastructure allows us to run at high speeds despite the increase in communication.

Breaking up functionality in isolated components directly improves reliability. Making components smaller allows us to better contain the effects of failures. Moreover, components can often be restarted transparently, so that a bug in IP, say, will not affect TCP. Our system recovers seamlessly from crashes and hangs in drivers, network filters, and most protocol handlers. Since the restarted component can easily be a newer or patched version of the original code, the same mechanism allows us to update *on the fly* many core OS components (like IP, UDP, drivers, packet filters, etc.).

The OS architecture and the current trend towards many-core hardware together allow, for the first time, an architecture that has the reliability advantages of multiserver systems and a performance approximating that of monolithic systems [7] even though there are many optimizations left to exploit. The price we pay is mainly measured in the loss of cores now dedicated to the OS. However, in this paper we assume that cores are no longer a scarce resource as high-end machines already have dozens of them today and will likely have even more in the future.

Outline: In Section II, we discuss the relation between reliability, performance, multiservers and multicores. Next, in Section III, we explain how a redesign of the OS greatly improves performance problems without giving up reliability. We present details of our framework in Section IV and demonstrate the practicality of the design on the networking stack in Section V. The design is evaluated in Section VI We compare our design to related work in Section VII and conclude in Section VIII.

II. Reliability, performance and multicore

Since it is unlikely that software will ever be free of bugs completely [19], it is crucial that reliable systems be able to cope with them. Often it is enough to restart and the bug disappears. For reliability, new multicore processors are double-edged swords. On the one hand, increasing concurrency leads to new and complex bugs. On the other hand, we show in this paper that the abundance of cores and a carefully designed communication infrastructure allows us to run OS components on dedicated cores—providing isolation and fault-tolerance without the performance problems that plagued similar systems in the past.

Current hardware trends suggest that the number of cores will continue to rise [2], [3], [23], [29], [35], [36] and that the cores will specialize [31], [39], [41], for example for running system services, single threaded or multithreaded applications. As a result, our view on processor cores is changing, much like our view on memory has changed. There used to be a time when a programmer would know and cherish every byte in memory. Nowadays, main memory is usually no longer scarce and programmers are not shy in wasting it if doing so improves overall efficiency—there is plenty of memory. In the same way, there will soon be plenty of cores. Some vendors already sacrifice cores for better energy efficiency [2], [3]. The key assumption in this paper is that *it is acceptable to utilize extra cores to improve dependability and performance*.

Unfortunately, increasing concurrency makes software more complex and, as a result, more brittle [24]. The OS is no exception [28], [32], [34]. Concurrency bugs lead to hangs, assertion failures, or crashes and they are also particularly painful, as they take considerably longer to find and fix than other types of bugs [24].

Thus, we observe (a) an increase in concurrency (forced by multicore hardware trends), (b) an increase in concurrency bugs (often due to complexity and rare race conditions), and (c) systems that crash or hang when any of the OS components crashes or hangs. While it is hard to prevent (a) and (b), we *can* build a more reliable OS that is capable of recovering from crashing or hanging components, whether they be caused by concurrency bugs or not.

Our design improves OS reliability both by *structural* measures that prevent certain problems from occurring in the first place, and by *fault recovery* procedures that allow the OS to detect and recover from problems. Structural measures include fault isolation by running OS components as unprivileged user processes, avoiding multithreading in components, and asynchronous IPC. For fault recovery, we provide a monitor that checks whether OS components are still responsive and restarts them if they are not.

The research question addressed in this paper is whether we can provide such additional reliability without jeopardizing performance. In existing systems, the answer would be: "No". After all, the performance of multiserver degrades quickly with the increase in IPC incurred by splitting up the OS in small components.

This is true even for microkernels like L4 that have optimized IPC greatly [26], and is the main reason for the poor performance of multiserver systems like MINIX 3 [1]. However fast we make the mechanism, kernel-based IPC *always* hurts performance: every trap to the kernel pollutes the caches and TLB with kernel *stuff*, flushes register windows, and messes up the branch predictors.

In the next section, we discuss how we can reduce this cost in a new reliable OS design for manycore processors.

III. Rethinking the Internals

As a first step, and prior to describing our design, we identify the tenets that underlie the system. Specifically, throughout this work we adhere to the following principles:

1) **Avoid kernel involvement on the fast path.** Every trap in the kernel pollutes caches and branch predictors and should be avoided when performance counts.

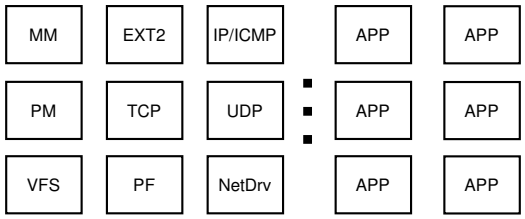

Figure 1. Conceptual overview of NewtOS. Each box represents a core. Application (APP) cores are timeshared.

2) **Do not share cores for OS components.** There is no shortage of cores and it is fine to dedicate cores to OS components, to keep the caches, TLBs and branch prediction warm, and avoid context switching overhead.
3) **Split OS functions in isolated components.** Multiple, isolated components are good for fault tolerance: when a component crashes, we can often restart it. Moreover, it allows us to update the OS on the fly.
4) **Minimize synchronous intra-OS communication.** Synchronous IPC introduces unnecessary waits. Asynchronous communication avoids such bottlenecks. In addition, asynchrony improves reliability, by preventing clients from blocking on faulty servers [21].

We now motivate the most interesting principles in detail.

A. IPC: What's the kernel got to do with it?

All functions in a multiserver system run in isolated servers. A crash of one server does not take down the entire system, but the isolation also means that there is no global view of the system and servers rely on IPC to obtain information and services from other components. A multiserver system under heavy load easily generates hundreds of thousands of messages per second. Considering such IPC rates, both the direct and indirect cost of trapping to the kernel and context switching are high.

To meet the required message-rate, we remove the kernel from high-frequency IPC entirely and replace it with trusted communication channels which allow fast asynchronous communication. Apart from initially setting up the channels, the kernel is not involved in IPC at all (Section IV).

As shown in Figure 1, every OS component in NewtOS can run on a separate core for the best performance while the remaining cores are for user applications. The OS components themselves are single-threaded, asynchronous user processes that communicate without kernel involvement. This means no time sharing, no context switching, and competing for the processor resources. Caching efficiency improves both because the memory footprint of each component is smaller than of a monolithic kernel, and because we avoid many expensive flushes. By dedicating cores to OS components, we further reduce the role of the kernel because no scheduling is required and dedicated cores handle interrupts. This leaves only a thin kernel layer on each system core.

Removing the kernel from fast-path IPC also removes the additional inefficiency of cross-core IPC that is, paradoxically, only noticeable because there is no longer a context switch. Single core systems partially hide IPC overhead behind the context switch. If a server needs another server to process a request, that process must be run first. Therefore, the trap to the kernel to send a message is the same as needed for the context switch, so some of the overhead is "hidden".

On multicores, context switching no longer hides the cost of IPC and the latency of the IPC increases because of the intercore communication. The kernel copies the message and if one of the communicating processes is blocked receiving, it must wake it up. Doing so typically requires an interprocessor interrupt which adds to the total cost and latency of the IPC.

If enough cores are available, we can exclude the kernel from the IPC. Our measurements show that doing so reduces the overhead of cross-core communication dramatically.

B. Asynchrony for Performance and Reliability

A monolithic system handles user space requests on the same core as where the application runs. Many cores may execute the same parts of the kernel and access the same data simultaneously, which leads to lock contention to prevent races and data corruption. We do not require CPU concurrency per server and event-driven servers are fast and arguably less complex than threads (synchronization, preemption, switching, etc.) and help avoid concurrency bugs. For us, single threaded servers are a good design choice.

However, synchronous communication between the servers (blocking until receiving a reply), as used in most multiserver systems may well make the entire system single threaded in practice. Thus, dedicating a separate core to each server reduces the communication cost but does not scale further.

Ideally, we would like the cores to process tasks in parallel if there is work to do. To do so, the servers must work as independently of each other as possible to increase intra-OS parallelism. Only asynchronous servers can process other pending requests while they wait for responses from others.

An argument against asynchrony is that it is difficult to determine whether a process is just slow or whether it is dead. However, a multiserver system, unlike a distributed system, runs on a single machine and can take advantage of fast and reliable communication provided by the interconnect. Together with the microkernel, it makes detection of such anomalies much simpler.

Most microkernels provide synchronous IPC because it is easy to implement and requires no buffering of messages. In practice, support for asynchronous communication is either inefficient (e.g., Minix 3) or minimal. Specifically, the large number of user-to-kernel mode switches results in significant slowdowns here also. In contrast, the communication channels in our design increase asynchrony by making nonblocking calls extremely cheap.

While asynchrony is thus needed for scalability on multicores, it is equally important for dependability. The system should never get blocked forever due to an unresponsive or

dead server or driver. In our design, a misbehaving server cannot block the system even if it hogs its core. Better still, our asynchronous communication lets servers avoid IPC deadlocks [38]. Since servers decide on their own from which channel and when to receive messages (in contrast to the usual *receive from anyone* IPC call), they can easily handle a misbehaving server that would otherwise cause a denial-of-service situation.

IV. Fast-Path Channels

The main change in our design is that instead of the traditional IPC mechanisms provided by the microkernel, we rely on asynchronous channels for all fast-path communication. This section presents details of our channel implementation using shared memory on cache-coherent hardware. Shared memory is currently the most efficient communication option for general-purpose architectures. However, there is nothing fundamental about this choice and it is easy to change the underlying mechanism. For instance, it is not unlikely that future processor designs will not be fully cache coherent, perhaps providing support for the sort of message passing instead as provided by the Intel SCC [23]. Moreover, besides convenient abstractions, our channels are generic building blocks that can be used throughout the OS. By wrapping the abstraction in a library, any component can set up channels to any other component.

Our channel architecture has three basic parts: (1) queues to pass requests from one component to another, (2) pools to share large data, and (3) a database containing the requests a component has injected in the channels and which we are waiting for to complete or fail. We also provide an interface to manage the channels. The architecture draws on FBufs [12] and Streamline [10], but is different from either in how it manages requests.

Queues: Each queue represents a unidirectional communication channel between one sender and one consumer. We must use two queues to set up communication in both directions. Each filled slot on a queue is a marshalled request (not unlike a remote procedure call) which tells the receiver what to do next. Although we are not bound by the universal size of messages the kernel allows and we can use different slot sizes on different queues, all slots on one queue have the same size. Thus we cannot pass arbitrarily sized data through these channels.

We use a cache friendly queue implementation [17], [10], that is, the head and tail pointers are in different cache lines to prevent them from bouncing between cores. Since the queues are single-producer, single-consumer they do not require any locking and adding and removing requests is very fast. For instance, on the test machine used in the evaluation section, the cost of trapping to the kernel on a single core using the `SYSCALL` instruction in a void Linux system call takes about 150 cycles if the caches are hot. The same call with cold caches takes almost 3000 cycles. In contrast, on our channels it requires as little as 30 cycles to asynchronously enqueue a message in a queue between 2 processes on different cores while the receiver keeps consuming the messages. The cost includes the stall cycles to fetch the updated pointer to the local cache.

Pools: We use shared memory pools to pass large chunks of data and we use rich pointers to describe in what pool and where in the pool to find them. Unlike the queues which are shared exclusively by the two communicating processes, many processes can access the same pool. This way we can pass large chunks from the original producer to the consumers further down the line without the need to copy. Being able to pass long chains of pointers and zero-copy are mechanism crucial for good performance. All our pools are exported read only to protect the original data.

Database of requests: As our servers are single-threaded and asynchronous, we must remember what requests we submitted on which channels and what data were associated with each request. After receiving a reply, we must match it to the corresponding request. For this purpose, the architecture provides a lightweight request database that generates a unique request identifier for every request.

Our channel architecture also provides an interface to publish the existence of the channels, to export them to a process, and to attach to them. We discuss this in more detail in Section IV-C.

A. Trustworthy Shared Memory Channels

Shared memory has been used for efficient IPC for a long time [12] and in many scenarios [11], [6], [10], [27]. The question we address here is whether we can use it as a trusted communication channel without harming dependability.

Kernel-level IPC guarantees that a destination process is reliably informed about the source process. Our channels offer the same guarantees. As servers must use the trusted (and slower) kernel IPC to set up the channels (requesting permission to export or attach to them), the kernel ensures that processes cannot change the mappings to access and corrupt other processes' address spaces. Since a process cannot make part of its address space available to another process all by itself, setting up the shared memory channel involves a third process, known as the virtual memory manager. Each server implicitly trusts the virtual memory manager. Once a shared memory region between two processes is set up, the source is known.

Likewise, we argue that communication through shared memory is as reliable as communication through the kernel. In case the source is malicious or buggy, it can harm the receiving process by changing data in the shared location when the receiver was already cleared to use them. The receiving process must check whether a request make sense (e.g., contains a known operation code) and ignore invalid ones. If the sender tampers with the payload data, the effect is the same as if it produced wrong data to begin with. Although

incorrect data may be sent to the network or written to disk, it does not compromise the network or disk driver.

In addition, we use write protection to prevent the consumer from changing the original data. While the consumer can, at any time, pass corrupted data to the next stage of a stack, if a request fails or we need to repeat the request (e.g., after a component crash, as discussed in Section V), we can always use the original data.

We must never block when we want to add a request and the queue is full, as this may lead to deadlocks. Each server may take its own unique action in such a situation. For instance, dropping a packet within the network stack is acceptable, while the storage stack should remember the request until the congestion is resolved.

B. Monitoring Queues

If a core is busy, there is no problem to check the queues for new requests. However, once a core is not fully loaded, constant checking keeps consuming energy, even though there is no work to do. Therefore, we put idle cores to sleep. But the process must wake up immediately when there is more work to do. The sender can use kernel IPC to notify the receiver that a new request is available, but that is precisely what we want to avoid. To break the circle, we use the `MONITOR` and `MWAIT` pair of instructions, recently added to the Intel x86 instruction set, to monitor writes to a memory location while the core is idle. In addition to the shared memory channels, each server exports the location it will monitor at idle time, so the producers know where to write to.

Unfortunately, these instructions are, available only in privileged mode—so we must use the kernel to sleep. Although we only need the kernel's assistance when a server has no work to do and we want to halt the core, the overhead of restoring the user context when a new request arrives adds to the latency of the `MWAIT`. This fact encourages more aggressive polling to avoid halting the core if the gap between requests is short. Part of the latency is absorbed by the queues we use to implement the communication channels. If the `MWAIT` were optionally allowed in unprivileged mode, we would get perfect energy consumption aware polling with extremely low wake-up latency. In our opinion, the kernel should be able to allow this instruction in an unprivileged mode (as it can disable it in the privileged one) when it knows that the core is dedicated to a process and thus this process cannot prevent other processes from running when it halts its core. Moreover, a core cannot be disabled entirely, as an interrupt, for example from another core, can always wake it up. Although such instructions are fairly unique to x86, they prove so useful that we expect other architectures to adopt variants of them in the future.

C. Channel Management

As there is no global manager in our system, the servers must set up the channels themselves. After all, we do not want our recovery mechanisms to depend on another server which itself may crash. When a server starts, it announces its presence by a publish-subscribe mechanism. Another server subscribed to the published event can then export its channels to the newly started one. Exporting a channel provides the recipient with credentials to attach to it. In our case, it can use the credentials to request the memory manager to map it into its address space. A server can also detach from a channel. This is only used when the other side of the channel disappears. We never take a channel away from an active server since it would crash after accessing the unmapped memory. Pools are basically channels without the additional queue structuring and the limit of how many processes can attach to it, therefore we use the same management for both.

Because we use the pools to pass large chunks of data without copying, not only the processes that communicate immediately with each other must be able to attach pools. Each channel is identified by its creator and a unique id. The creator publishes the id as a key-value pair with a meaningful string to which a server can subscribe. After obtaining the identification pair, the server can request an export of the pool from its creator, which the creator can grant or deny.

D. Channels and Restarting Servers

When a server crashes and restarts it has to reattach channels which were previously exported to it. Since the channels exported by a crashed server are no longer valid, their users need to detach from them and request new exports. The identification of the channels does not change.

We cannot hide the fact that a server crashed from the ones it talked to since there may have been many requests pending within the system. Reestablishing the channels to a server which recovered from a crash is not enough. Servers that kept running cannot be sure about the status of the requests they issued and must take additional actions. We use the request database to store each request and what to do with it in such a situation. We call this an *abort* action (although a server can also decide to reissue the request). When a server detects a crash of its neighbor, it tells the database to abort all requests to this server. While the database removes the requests, it executes the associated abort actions. Abort actions are highly application specific. For instance, servers in a storage stack are likely to clean up and propagate the abort further until an error is returned to the user-space processes. On the other hand, servers in a networking stack may decide to retransmit a packet or drop it, which we discuss in the following Section V.

The channels allow a component to be transparently replaced by a newer version on the fly as long as the interface to the rest of the system stays unchanged. Since a new

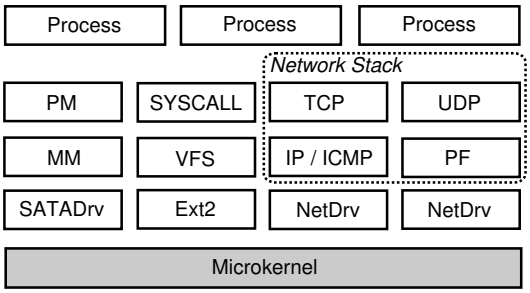

Figure 2. Decomposition and isolation in multiserver systems

incarnation of a server in our system inherits the old version's address space, the channels remain established.

V. Dependable Networking Stack

The network stack is a particularly critical part of current OSs, where often extreme performance is as important as high reliability since downtime may have a high cost. In addition, the network stack is very complex and frequently contains critical bugs, as witnessed recently by the vulnerability in Microsoft systems [4]. Thus, we selected the networking stack as the most interesting subsystem in the OS to evaluate our design.

In contrast to monolithic OSs that are very good in performance but do not address reliability at all, we opted for an extreme design. In case of a fatal error in the stack, the rest of the system keeps working. As we shall see, the system can often fix the problem automatically and seamlessly. In situations when it cannot, the user can take an action like saving data to disk and reboot which is still more than a user can do when the whole system halts.

Our stack goes even well beyond what is currently found in other multi-server systems. For instance, Herder et al. [22] showed in the original Minix 3 how to repair faulty network userspace drivers at runtime by restarting them. However, network drivers are near-stateless components and the network protocols know how to recover from packet loss. Any fault in IP, say, would crash the entire stack. However, because the network stack itself is stateful, it was possible to restart it, but not to recover the state. We decompose the network stack in even more smaller (and simpler) separate processes, which increases isolation, but also the amount of IPC.

Figure 2 shows how we split up the stack into multiple components. The dashed box represents what is usually a single server in a multiserver system and the boxes inside are the servers in NewtOS. We draw a line between the IP layer and the transport protocols. Our IP also contains ICMP and ARP. For security reasons, the networking stack usually contains a packet filter which we can also isolate into a standalone process. Again, such an extreme decomposition is practical only if we do not significantly compromise performance.

Each of the components has at least some state and the size of this state and the frequency at which it changes determines how easily we can recover from a failure (Table I).

Component	Ability to restart
Drivers	No state, simple restart
IP	Small static state, easy to restore
UDP	Small state per socket, low frequency of change, easy to store safely
Packet filter	Static configuration, easy to restore, information about existing connections is recoverable
TCP	Large, frequently changing state for each connection, difficult to recover. Easy to recover listening sockets

Table I
Complexity of recovering a component

After drivers, the simplest component to restart is IP. It has very limited (static) state, basically the routing information, which we can save in any kind of permanent storage and restore after a crash. ARP and ICMP are stateless. To recover UDP, however, we need to know the configuration of the sockets, a 4-tuple of source and destination address and ports. Fortunately, this state does not change very often. The packet filter has two kinds of state. The more static portion is its configuration by the user which is as simple to recover as IP state. However, there is also dynamic state. For instance, when a firewall blocks incoming traffic it must not stop data on established outgoing TCP connections after a restart. In NewtOS, the filter can recover this dynamic state, for instance, by querying the TCP and UDP servers.

The biggest challenge is recovering TCP. Besides the 4-tuple part of the state, it has a frequently changing part for congestion avoidance and reliable transport. In fact, all unacknowledged data are part of this state. Although preserving such state for recovery is difficult, research in this area shows how to design such system components [9].

In our design, we isolate the parts that are difficult to recover (TCP) from from those we know how to restart, thus improving overall dependability. The ability to recover most of the network stack (even if we cannot recover all) is much better than being able to recover none of it and vastly better than a bug bringing the entire system to a grinding halt. Note that not being able to recover the state of TCP means only that existing connections break. Users can immediately establish new ones.

NewtOS survives attacks similar to the famous ping of death [14] without crashing the entire system. Also, it does not become disconnected when the packet filter crashes, neither does it become vulnerable to attacks after it restarts since its configuration is preserved.

In addition, it is possible to update each component independently without stopping the whole system as long as the interface to the rest of the system remains unchanged. In fact, shutting down a component gracefully makes restarting much simpler as it can save its state and announce the restart to other parts of the stack in advance. We are confident that all servers of our network stack can converge to a consistent

state for an update since they satisfy the conditions presented by Giuffrida et al. in [18].

In November 2011, Microsoft announced a critical vulnerability [4] in the UDP part of Windows networking stack. The vulnerability allows an intruder to hijack the whole system. In this respect, NewtOS is much more resilient. First, hijacking an unprivileged component does not automatically open doors to the rest of the system. Second, we are able to *replace* the buggy UDP component without rebooting. Given the fact that most Internet traffic is carried by the TCP protocol, this traffic remains completely unaffected by the replacement, which is especially important for server installations. Incidentally, restartability of core components proved very valuable during development of the system since each reboot takes some time and it resets the development environment.

A. The Internals

Nowadays, multigigabit networks present a challenge for many software systems, therefore we want to demonstrate that a multiserver system handles multigigabit rates. We replaced the original Minix 3 stack by lwIP [13] because lwIP is easier to split and modify. Although lwIP is primarily designed for size rather than high performance (it targets mostly embedded devices), it is a clean and portable implementation of the TCP/IP protocol suite. We use the NetBSD packet filter (PF) and we heavily modified the driver for the family of Intel PRO/1000 gigabit network adapters.

To separate the IP part, we only had to change the place where lwIP does the routing for outgoing packets. Although virtually all gigabit network adapters provide checksum offloading and TCP segmentation offloading (TSO - NIC breaks one oversized TCP segment into small ones), lwIP does not support it out of the box. We changed the lwIP internals to support these optimizations. Although this improves the performance of lwIP dramatically, the TCP code requires a complete overhaul if we want it to be as efficient as, say, the Linux network stack. Even so, we will show that the performance of our design is competitive.

We did not port the network stack from Linux or any BSD flavor because these depend on the monolithic environment (memory management, etc.) and changing the stack to our needs would likely severely compromise its performance.

Figure 3 shows the placement of PF within the stack. Placing PF in a *T* junction makes it easier to support both post and pre-routing rules, and to restart PF on a crash (see Section V-D). In addition, in this design IP remains the only component that communicates with drivers. Although this setup puts more performance pressure on the IP server since it must hand off each packet to another component three times, IP is not the performance bottleneck of the stack, even with the extra work.

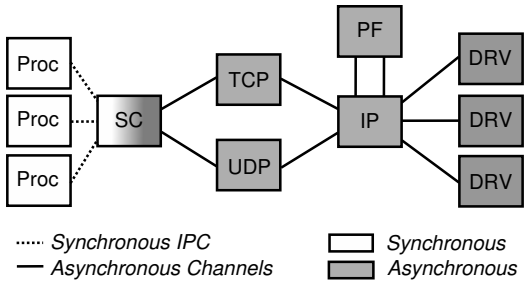

Figure 3. Asynchrony in the network stack

B. Combining Kernel IPC and Channels IPC

In our current implementation, the servers which interface with user space and drivers need to combine channel IPC with kernel IPC, as the kernel converts interrupts to messages to the drivers. Similarly, the system calls from user applications are also kernel IPC messages. Therefore we combine the kernel call, which monitors memory writes, with a non-blocking receive, that is, before we really block or after we wake up, we check if there is a pending message. Whenever there is one, we deliver it when we return from the kernel call. Of course, we do not block at all if we find a message. Because kernel IPC from other cores is accompanied by an interprocessor interrupt (IPI) when the destination core is idle, the IPI breaks the memory write monitoring even if no write to the monitored location occurred. Note that unlike in a monolithic design where system calls are kernel calls, system calls in a multiserver system are implemented as messages to servers.

To detach the synchronous POSIX system calls from the asynchronous internals of NewtOS, the applications' requests are dispatched by a SYSCALL server. It is the only server which frequently uses the kernel IPC. Phrased differently, it pays the trapping toll for the rest of the system. Nonetheless, the work done by the SYSCALL server is minimal, it merely peeks into the messages and passes them to the servers through the channels. The server has no internal state, and restarting it in the case of a failure is trivial. We return errors to the system calls and ignore old replies from the servers. Figure 3 shows connections of the SYSCALL (SC) server to the rest of the network stack. We use these connections only for control messages. The actual data bypass the SYSCALL as opening a socket also exports shared memory buffer to the applications where the servers expect the data.

Our C library implements the synchronous calls as messages to the SYSCALL server, which blocks the user process on receive until it gets a reply. Although this is a convenient way to implement POSIX system calls, some applications may prefer other arrangements. Extending the channels from inside the system to the user space allows applications to bypass the overhead of the synchronous calls by opening channels directly to the servers.

C. Zero Copy

By using channels, shared pools and rich pointers, we can pass data through the system without copying it from component to components as is traditionally done in multiservers. Any server that knows the pool described in the pointer, can translate the rich pointer into a local one to access the data.

Because modern network interface cards (NICs) assemble packets from chunks scattered in memory, monolithic systems pass packets from one networking layer to another as a chain of these chunks. Every protocol prepends its own header. The payload is similarly scattered, especially when the packets are large (for example, when the network allows jumbo frames or TSO is enabled). In NewtOS, we pass such a chain as an array allocated in a shared pool filled with rich pointers.

We emphasize that zero copy makes crash recovery much more complicated. Unlike a monolithic system where we can free the data as soon as we stop using them, in our case, the component that allocated the data in a pool must free them. This means that we must report back when it is safe to free the data—almost doubling the amount of communication. Worse, after a server recovers from a crash, the other servers must find out what data are still in use and which should be freed. To the best of our knowledge, ours is the first system capable of restarting components in a multiserver system stack with zero copy communication throughout.

To further improve reliability, we make the data in the pools immutable (like in FBufs [12]). Phrased differently, we export all pools read-only. Therefore each component which needs to change data must create a new copy. For instance, this is done by IP when it places a partial checksum in the TCP and UDP headers of outgoing packets. As the headers are tiny, we combine them with IP headers in one chunk.

D. Crash Recovery

Before we can recover from a crash, we must detect it. In NewtOS, as in Minix 3, all system servers are children of the same reincarnation server which receives a signal when a server crashes, or resets it when it stops responding to periodic heartbeats. More details on crash detection in Minix 3 are presented in [20].

A transparent restart is not possible unless we can preserve the server's state and we therefore run a storage process dedicated to storing interesting state of other components as key and value pairs. We start each server either in *fresh start* or in *restart* mode so the process knows whether it should try to recover its state or not. It can request the original state from the storage component. If the storage process itself crashes and comes up, every other server has to store its state again.

Recovering from a crash of other components is very different. When a system component crashes and restarts, it must tell everyone it wants to talk to that it is ready to set up communication channels and to map whatever pools it needs. At that point, its neighbors must take action to discover the status of requests which have not completed yet. All the state a component needs to restart should be in the storage server.

Drivers: State of the art self-healing OSs, like Minix 3, previously demonstrated restarting of simple network drivers [22], but it feeds only a single packet to a driver at a time. In contrast, we asynchronously feed as much data as possible to be able to saturate multigigabit links and use more complex features of the hardware. In addition, our drivers do not copy the packets to local buffers.

As a result, the IP server must wait for an acknowledgment from the driver that a packet was transmitted before it is allowed to free the data. IP knows which packets were not yet accepted by the driver for processing from the state of the queue. It is likely that all packets except the last one were successfully transmitted, but the last one (as the driver perhaps crashed while processing it). Although network protocols are designed to deal with lost packets, we do not want to drop more than necessary. In case of doubt, we prefer to send a few duplicates which the receiver can decide to drop. Therefore IP resubmits the packets which it thinks were not yet transmitted.

A faulty driver may make the device operate incorrectly or stop working at all. This can be also a result of differences between specification and implementation of the hardware. It is difficult to detect such situations. When we stop receiving packets, it can either be because nobody is sending anything, or because the device stopped receiving. As a remedy, we can detect that a driver is not consuming packets for a while or that we do not receive replies to echo packets and then restart the driver pro-actively. However, these techniques are out of the scope of this paper and unless a driver crashes, we can not currently recover from such situations.

IP: To recover the IP server, it needs to store its configuration, IP addresses of each device and routing like the default gateway, etc. This information changes rarely on the network edge. Because IP allocates a pool which the drivers use to receive packets, the drivers must make sure that they switch these pools safely, so the devices do not DMA to freed memory. It turned out that we must reset the network cards since the Intel gigabit adapters do not have a knob to invalidate its shadow copies of the RX and TX descriptors. Therefore a crash of IP means defacto restart of the network drivers too. We believe that restart-aware hardware would allow less disruptive recovery.

Similarly, TCP and UDP may have packets still allocated in the old receive pool and they must keep a reference to it until all the packets are delivered or discarded. On the other hand, neither can free the transmitted packets until they know that no other component holds a reference to the memory. Our policy in both cases is that we resubmit the requests to IP. We generate new identifiers so that we can ignore replies to the original requests and only free the data once we get replies to the new ones. This also means that we may transmit some duplicates. However, in case of TCP, it is much

more important that we quickly retransmit (possibly) lost packets to avoid the error detection and congestion avoidance. This helps to recover quickly the original performance after recovering the functionality.

UDP: The UDP server saves in the storage server which sockets are currently open, to what local address and port they are bound, and to which remote pair they are connected (preset for `send`). It is easy to recreate the sockets after the crash. However, UDP does not store whether a process was blocked on a socket operation and if so, which one (and doing so would result in significant overhead). On the other hand, the SYSCALL server remembers the last unfinished operation on each socket and can issue it again. This is fine for `select` and `recv` variants as they do not trigger any network traffic. In contrast, `send` variants will result in packets sent. As mentioned previously, we tend to prefer sending extra data. Of course, we can also return an error to the application instead, for example, zero bytes were written.

TCP: Much like UDP, TCP also saves in the storage server the sockets that are open. In addition, TCP saves in what state the connection is (listening, connecting, established, etc.) so the packet filter can restore connection tracking after its crash. TCP can only restore listening sockets since they do not have any frequently changing state and returns error to any operation the SYSCALL server resubmits except `listen`.

Packet filter: To restore the optional packet filter we need to recover the configuration (much like restoring IP configuration) and the open connections (much like restoring TCP or UDP sockets) and it stores this information in the storage server. Since IP must get a reply for each request before it can pass a packet further the stack, it can safely resubmit all unfinished requests without packet loss and generating duplicate traffic.

VI. Evaluation

We evaluate our multiserver design and present the benefits of the principles we are advocating for. To demonstrate the competitiveness of our design, we evaluate on a 12 core AMD Opteron Processor 6168 (1.9GHz) 4GB RAM with 5 Intel PRO/1000 PCI Express gigabit network adapters. We are limited by the number of PCIe slots in our test machine. We use standard 1500 byte MTU in all configurations.

A. TCP Performance

Table II shows peak performance of our TCP implementation in various stages of our development along with original Minix 3 and Linux performance. The table is ordered from the least performing at the top to the best performing at the bottom. The first line shows that a fully synchronous stack of Minix 3 cannot efficiently use our gigabit hardware, on the other hand, line 4 shows that a single server stack which adopts our asynchronous channels can saturate 4 of our network interfaces and more with additional optimizations

1	Minix 3, 1 CPU only, kernel IPC and copies	120Mbps
2	NewtOS, Split stack, dedicated cores	3.2Gbps
3	NewtOS, Split stack, dedicated cores + SYSCALL	3.6Gbps
4	NewtOS, 1 server stack, dedicated core + SYSCALL	3.9Gbps
5	NewtOS, 1 server stack, dedicated core + SYSCALL + TSO	5+Gbps
6	NewtOS, Split stack, dedicated cores + SYSCALL + TSO	5+Gbps
7	Linux, 10Gbe interface	8.4Gbps

Table II
Peak performance of outgoing TCP in various setups

(line 5). Line 3 presents the advantage of using the SYSCALL server, in contrast to line 2, to decouple synchronous calls from asynchronous internals. Comparing lines 3 and 4, we can see the effect of communication latency between the extra servers in the split stack. Using TSO we remove a great amount of the communication and we are able to saturate all 5 network cards while allowing parts of the stack to crash or be live-updated. It is important to mention that Linux also cannot saturate all the devices without using TSO which demonstrates that not only the architecture of the stack but also its ability to offload work to network cards and reduction of its internal request rate (TCP window scaling option, jumbo frames, etc.) play the key role in delivering the peak performance. To put the performance in perspective, line 7 shows the maximum we obtained on Linux on the same machines with standard offloading and scaling features enabled using a 10Gbe adapter which neither Minix 3 or NewtOS support.

We carried out our experiments with one driver per network interface, however, to evaluate scalability of the design we also used one driver for all interfaces, which is similar to having one multi-gigabit interface. Since the work done by the drivers is extremely small (filling descriptors and updating tail pointers of the rings on the device, polling the device) coalescing the drivers into one still does not lead to an overload. In contrary, the busy driver reduces some latency since it is often awake and ready to respond.

We believe that on a heavily threaded core like that of Oracle Sparc T4, we would be able to run all the drivers on a single core using the threads as the containers in which the drivers can block without sacrificing more cores and still delivering the same performance and isolation of drivers.

B. Fault Injection and Crash Recovery

To assess the fault tolerance of our networking stack we have injected faults in the individual components. Therefore we used a fault injection tool equal to that used by the authors of Rio file cache [30], Nooks [42] and Minix 3 [22] to evaluate their projects. We summarize the distribution of the faults in Table III and effects the crashes have in Table IV. During each test run we injected 100 faults into a randomly selected component. When the component did not crash within a minute we rebooted the machine and continued

Total	TCP	UDP	IP	PF	Driver
100	25	10	24	25	16

Table III
DISTRIBUTIONS OF CRASHES IN THE STACK

Fully transparent crashes	70
Reachable from outside	90 (+ 6 manually fixed)
Crash broke TCP connections	30
Transparent to UDP	95
Reboot necessary	3

Table IV
CONSEQUENCES OF A CRASHES

with another run. We collected 100 runs that exhibited a crash and we observed the damage to the system. While injecting the faults we stressed the components with a TCP connection and periodic DNS queries. The tool injects faults randomly so the faults are unpredictable. Since some of the code does not execute during a normal operation and because of different fraction of active code, some components are more likely to crash than the others.

The most serious damage happens when the TCP server crashes. In these cases all established connections disappear. On the other hand, since we recover sockets which listen for incoming connections, we are able to immediately open new ones to our system. We used *OpenSSH* as our test server. After each crash we tested whether the active ssh connections kept working, whether we were able to established new ones and whether the name resolver was able to contact a remote DNS server without reopening the UDP socket.

We were able to recover from vast majority of the faults, mostly transparently. After the 25 TCP crashes, we where able to directly reconnect to the SSH server in 19 of those cases. In 3 of the cases we had to manually restart the TCP component to be able to reconnect to the machine. In two other cases a faulty IP and a not fully responsive driver was the reason why it was impossible to connect to the machine. Manually restarting the driver respectively IP solved the problem. In three cases we had to reboot the system due to hangs in the synchronous part of the system which merges sockets and file descriptors for `select` and has not been modified yet to use the asynchronous channels we propose. This suggests that reliability of other parts of the system would also greatly benefit from our design changes. In two cases, faults injected into a driver caused a significant slowdown but no crash. It is very likely that the faults misconfigured the network cards since the problem disappeared after we manually restarted the driver, which reseted the device.

In contrast to a solid production quality systems like Linux or Windows, NewtOS is a prototype and we do not have an automated testing infrastructure and thus had to run the fault injection tests manually. Therefore we were not able to make statistically significant number of runs. However, the results correlate with our expectations.

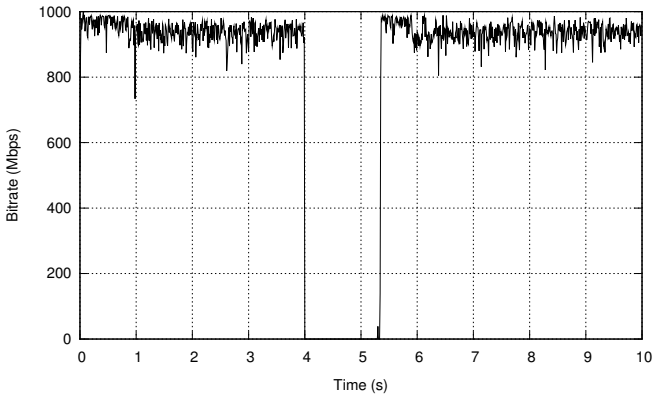

Figure 4. IP crash

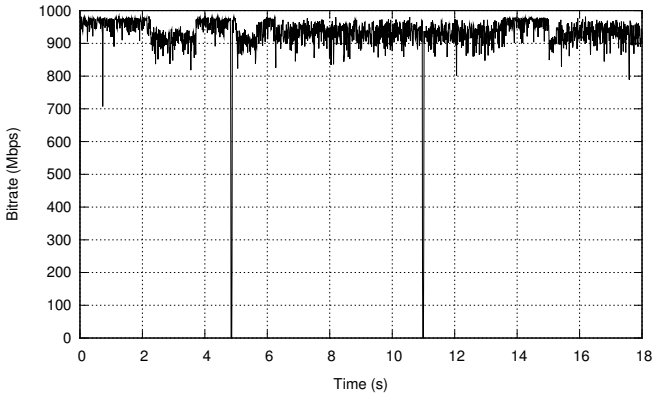

Figure 5. Packet filter crash

C. High Bitrate Crashes

A random crash during a high frequency of operations can cause a serious damage to the network traffic due to a loss of many packets. Figure 4 presents a bitrate sample of a connection between *iperf* on NewtOS and on Linux. We used *tcpdump* to capture the trace and *Wireshark* to analyze it. Using a single connection allows us to safely capture *all* packets to see all lost segments and retransmission. The trace shows a gap when we injected a fault in the IP server 4s after the start of the connection. We did not observe any lost segments and only one retransmission from the sender (due to a missing ACK and a timeout) which has been already seen by the receiver. The connection quickly recovered its original bitrate. As we already mentioned before, due to the hardware limitations, we have to reset the network card when IP crashes. This causes the gap as it takes time for the link to come up again and so the driver. Therefore, all the traces we inspected after a driver crash look very similar.

A similar sample trace on Figure 5 shows that a packet filter (PF) crash is almost not noticeable. Due to our design, we never lose packets because IP must see a reply from PF, otherwise it knows that the packet was not processed. The trace shows two crashes and immediate recovery to the original maximal bitrate while recovering a set of 1024 rules.

VII. Related work

Our work is based on previous research in operating systems and it blends ideas from other projects with our own into a new cocktail. Although the idea of microkernel-based multiserver systems is old, historically they could not match the performance of the monolithic ones because they were not able to use the scarce resources efficiently to deliver matching performance. The current multicore hardware helps to revive the multiserver system idea. In a similar vein, virtual machines (invented in the 1960s) have recently seen a renaissance due to new hardware.

Monolithic systems, in their own way, are increasingly adopting some of the multiserver design principles. Some drivers and components like file systems can already run mostly in user space with kernel support for privileged execution. In addition, kernel threads are similar to the independent servers. The scheduler is free to schedule these threads, both in time and space, as it pleases. The kernel threads have independent execution context in the privileged mode and share the same address space to make data sharing simple, although they require locks for synchronization. Parts of the networking stack run synchronously when executing system calls and partly asynchronously in kernel threads, which may execute on different cores than the application which uses it, depending on the number and usage of the cores. Coarse grained locking has significant overhead; on the other hand, fine grained locking is difficult to implement correctly.

Interestingly, to use the hardware more efficiently, the kernel threads are becoming even more distinct from the core kernel code; they run on dedicated cores so as not to collide with the execution of user applications and with each other. An example is FlexSC's [39], [40] modification of Linux. It splits the available cores into ones dedicated to run the kernel and ones to run the applications. In such a setup, the multithreaded applications can pass requests to the kernel asynchronously and exception free which reduces contention on some, still very scarce, resources of each core.

Yet another step towards a true multiserver system is the IsoStack [37], a modification of AIX. Instances of the whole networking stack run isolated on dedicated cores. This shows that monolithic systems get a performance boost by dedicating cores to a particularly heavy task with which the rest of the system communicates via shared memory queues. Thus it is certainly a good choice for multiserver systems which achieve the same merely by pinning a component to a core without any fundamental design changes. The primary motivation for these changes is performance and they do not go as far as NewtOS, where we split the network stack into multiple servers. In contrast, our primary motivation is dependability and reliability while the techniques presented in this paper allow us to also achieve competitive performance.

We are not the first to partition the OS in smaller components. Variants less extreme than multi-server systems isolate a smaller set of OS components in user-level processes—typically the drivers [15], [25]. Barrelfish [5] is a so-called multikernel, microkernel designed for scalability and diversity, which can serve as a solid platform for a multiserver system. We are planning to port our network stack on top or it.

Hypervisors are essentially microkernels which host multiple isolated systems. Colp et al. [8] show how to adopt the multiserver design for enhanced security of Xen's Dom0. Periodic microreboots and isolation of components reduces its attack surface.

It is worth mentioning that all the commercial systems that target safety and security critical embedded systems are microkernel/multiserver real-time operating systems like QNX, Integrity or PikeOS. However, all of them are closed-source proprietary platforms therefore we do not compare to them. Unlike NewtOS, they target very constrained embedded environments, whereas we show that the same basic design is applicable to areas where commodity systems like Windows or Linux dominate.

VIII. Conclusion and Future Work

In this paper we present our view on future dependable operating systems. Our design excludes the kernel from IPC and promotes asynchronous user space communication channels. We argue that multiserver systems must distribute the operating system itself to many cores to eliminate its overheads. Only such asynchronous multiserver systems, in which each component can run whenever it needs to, will perform well while preserving their unique properties of fault resilience and live-updatability.

We describe the challenges of designing the system and present an implementation of a networking stack built on these principles. The amount of communication and data our stack handles as a result of high networking load suggests that our design is applicable to other parts of the system.

We admit that we loose many resources by dedicating *big* cores of current mainstream processors to system components and it must be addressed in our future work. We need to investigate how to efficiently adapt the system to its current workload, for instance by coalescing lightly utilized components on a single core and dedicating cores to heavily used ones. Equally importantly we are interested in how can we change future chips to match our needs the best. For example, can some of the *big* cores be replaced by many simpler ones to run the system?

Acknowledgments

This work has been supported by the European Research Council Advanced Grant 227874. We would like to thank Arun Thomas for his priceless comments on early versions of this paper.

REFERENCES

[1] Minix 3, Official Website and Download. http://www.minix3.org.

[2] big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2011.

[3] Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. Whitepaper - http://www.nvidia.com/, 2011.

[4] Vulnerability in TCP/IP Could Allow Remote Code Execution. http://technet.microsoft.com/en-us/security/bulletin/ms11-083, Nov. 2011.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. Proc. of Symp. on Oper. Sys. Principles, 2009.

[6] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *Proc. of Symp. on Oper. Sys. Des. and Impl.*, 2004.

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. of Symp. on Oper. Sys. Des. and Impl.*, 2010.

[8] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proc. of Symp. on Oper. Sys. Principles*, 2011.

[9] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: improving reliability through operating system structure. In *Proc. of Symp. on Oper. Sys. Des. and Impl.*, 2008.

[10] W. de Bruijn, H. Bos, and H. Bal. Application-Tailored I/O with Streamline. *ACM Transacations on Computer Systems*, 29, May 2011.

[11] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proc. of Sys. Admin. and Net. Engin. Conf.*, 2004.

[12] P. Druschel and L. L. Peterson. Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *Proc. of Symp. on Oper. Sys. Principles*, 1993.

[13] A. Dunkels. Full TCP/IP for 8-bit architectures. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[14] J. Erickson. *Hacking: The Art of Exploitation*. No Starch Press, 2003.

[15] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A New Architecture for Device Drivers. In *Workshop on Hot Top. in Oper. Sys.*, 2007.

[16] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. of workshop on Beyond the PC: new challenges for the oper. sys.*, 2000.

[17] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *PPoPP*, 2008.

[18] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. We Crashed, Now What? In *HotDep*, 2010.

[19] L. Hatton. Reexamining the Fault Density-Component Size Connection. *IEEE Softw.*, 14, March 1997.

[20] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure Resilience for Device Drivers. In *Proc. of Int. Conf. on Depend. Sys. and Net.*, 2007.

[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Countering IPC Threats in Multiserver Operating Systems (A Fundamental Requirement for Dependability). In *Pacific Rim Int. Symp. on Dep. Comp.*, 2008.

[22] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault Isolation for Device Drivers. In *Proc. of Int. Conf. on Depend. Sys. and Net.*, 2009.

[23] Intel. Single-Chip Cloud Computer. http://techresearch.intel.com/ProjectDetails.aspx?Id=1.

[24] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. RADBench: A Concurrency Bug Benchmark Suite. In *HotPar'11*, May 2011.

[25] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Gtz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level Device Drivers: Achieved Performance. *Computer Science and Technology*, 20, 2005.

[26] J. Liedtke, K. Elphinstone, S. Schönberg, H. Hrtig, G. Heiser, N. Islam, and T. Jaeger. Achieved ipc performance (still the foundation for extensibility), 1997.

[27] J. Löser, H. Härtig, and L. Reuther. A Streaming Interface for Real-Time Interprocess Communication. In *Workshop on Hot Top. in Oper. Sys.*, 2001.

[28] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41:103–116, October 2007.

[29] T. Mattson. Intel: 1,000-core Processor Possible. http://www.pcworld.com/article/211238/intel_1000core_processor_possible.html, Nov. 2010.

[30] W. T. Ng and P. M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999.

[31] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proc. of Symp. on Oper. Sys. Principles*, 2009.

[32] M. Peloquin, L. Olson, and A. Coonce. Simultaneity safari: A study of concurrency bugs in device drivers. University of WisconsinMadison Report, pages.cs.wisc.edu/~markus/736/concurrency.pdf, 2009.

[33] D. C. Sastry and M. Demirci. The QNX Operating System. *Computer*, 28, November 1995.

[34] M. Scondo. Concurrency and race conditions in kernel space (linux 2.6). *LinuxSupport.com (extract from "Linux Device Drivers")*, December 2009.

[35] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27, August 2008.

[36] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-treaded, Power-efficient, SPARC SOC. In *ASSCC'07* .

[37] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proc. of USENIX Annual Tech. Conf.*, 2010.

[38] J. S. Shapiro. Vulnerabilities in Synchronous IPC Designs. In *Proc. of IEEE Symp. on Sec. and Priv.* IEEE Computer Society, 2003.

[39] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of Symp. on Oper. Sys. Des. and Impl.*, 2010.

[40] L. Soares and M. Stumm. Exception-less System Calls for Event-Driven Servers. In *Proc. of USENIX Annual Tech. Conf.*, 2011.

[41] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen. Fast Switching of Threads Between Cores. *SIGOPS Oper. Syst. Rev.*, 43, April 2009.

[42] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. of Symp. on Oper. Sys. Principles*, pages 207–222, 2003.

[43] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proc. of Symp. on Cloud Computing*, 2010.