Towards a Scalable Multiprocessor User-level Environment

Udo Steinberg
Technische Universität Dresden
udo@hypervisor.org

Bernhard Kauer
Technische Universität Dresden
bk@vmmon.org

Abstract

Our previous research on NOVA has shown that a small and efficient virtualization environment can be built by using microkernel construction principles in the design of a virtualization environment [9]. In NOVA, virtual-machine monitors, device drivers and other system services run as user-level applications. The microhypervisor provides communication mechanisms and enforces temporal and spatial separation between the components running on top of it.

Due to the increasing numbers of cores on a chip, even in embedded systems, support for multiple processors should be a primary objective in the design of any new user-level environment. In this paper we show how to construct a scalable multiprocessor user-level environment that requires only a minimal set of kernel abstractions.

1. Motivation

NOVA is a virtualization architecture that consists of a small microhypervisor and a user-level environment running on top of it. The microhypervisor enforces spatial and temporal separation and implements only basic mechanisms for virtualization, communication and resource management.

User-level components provide additional operating system functionality. One type of component is the user-level virtual-machine monitor, which manages the execution of an unmodified guest operating system in a virtual machine. In our current implementation, each virtual machine has its own associated VMM and every virtual CPU in a virtual machine has a dedicated handler thread in the VMM. Therefore, the VMM can handle most VM exits of different virtual CPUs in parallel. However, when a virtual CPU accesses an emulated device, the VMM may need to contact the driver for the hardware device to submit or request data.

The hardware devices of the platform are managed by user-level device drivers that run in separate address spaces and are therefore isolated from the rest of the system. Each device driver acts as a server and exports a communication interface to the rest of the system. Client applications such as the VMM communicate with the device driver through its external interface in order to use the hardware device.

The decomposed user-level environment improves the dependability of the whole system because it ensures fault containment. If one component crashes, unrelated parts of the system remain unaffected. In some cases the faulty component can be restarted transparently. In contrast to monolithic systems where the whole operating system is implemented as a single entity in which all subsystems ultimately trust each other and communication between subsystems is unrestricted, the communication and trust relationships of a decomposed multiserver user environment can be much more fine-grained. However, the decomposed design introduces additional communication overhead between components. Instead of using direct function calls, components of the user-level environment must use interprocess communication to cross address-space boundaries.

The NOVA microhypervisor has been designed for low communication overhead and efficient multi-processor operation. The majority of operations on different CPUs can execute in parallel without the need for expensive synchronization or cross-processor signaling. User applications can schedule their threads on different physical processors and thereby fully exploit the parallelism of the underlying hardware platform. However, the performance of the system depends not just on the parallelism of individual applications, but also on the scalability of system services, such as file systems, network stacks and device drivers. If these components are implemented as single-threaded servers, service invocation can become a serious scalability bottleneck. The logical conclusion is to build all critical system components as multi-threaded servers. In this paper we explore the design space for the construction of scalable user-level components. We discuss what their external interface should look like and how servers can internally synchronize access to critical sections. This paper makes the following research contributions:

- We describe communication and synchronization primitives that facilitate the construction of scalable user-level components.

IIDS'10, April 13, 2010, Paris, France.
Copyright © 2010 ACM TBD…$10.00

- We analyze what functionality a multiprocessor kernel should provide to expose the full parallelism of the underlying hardware platform to user-level programs.
- We discuss how synchronous requests and asynchronous responses can be combined to achieve temporal separation and low communication latency.

2. Design

Scalability

When multiple clients access a single-threaded server, the server can handle only one request at a time and subsequent requests must wait for the previous request to finish. Because single-threaded servers handle all requests serially, the scalability of the system is limited. Amdahl's Law states that if P is the proportion of a program that can execute in parallel and $1 - P$ is the proportion that executes serially, the maximum speedup that can be achieved by using N processors is

$$\frac{1}{1 - P + \frac{P}{N}}$$

Figure 1 illustrates that a workload with a parallel portion of 95% only achieves a speedup of 10x with 16 processors. Furthermore, the maximum speedup is limited to 20x. A workload with a parallel portion of 90% achieves a speedup of at most 10x, no matter how many processors are used.

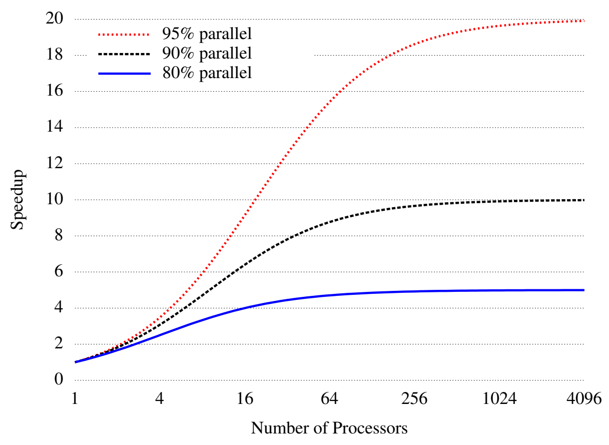

Figure 1: Amdahl's Law: The maximum speedup of a program is limited by the proportion of its parallel and serial parts.

To mitigate the impact of Amdahl's Law, our design goal is to minimize the amount of code that executes serially in each server of the user-level environment. Moreover, if a server needs to contact other servers to handle a client request, the length of the serial section is extended by the duration of all nested requests. Additionally, nested requests to other servers may have to wait for previous requests to those other servers to finish, which aggravates the scalability problem. We therefore construct system services as multi-threaded components. The number of concurrent requests that a server can handle now only depends on the number of worker threads provided by the server and the number of CPUs in the system.

Synchronization

A single-threaded server does not need internal synchronization, because the external interface serializes all execution within the server. In contrast, a multi-threaded server requires a synchronization mechanism that facilitates the implementation of critical sections. One example of a server-side critical section is access to device registers in a driver. Most devices require more than one register access to submit a request. Therefore, in a driver with multiple worker threads, only a single thread should program the hardware registers at any point in time.

Accounting

We also consider resource accounting to be an important design issue. When a server works on behalf of a client, the time spent handling the request can be either accounted to the client or to the server. Things get more complicated when more than one server is involved in the handling of a client request. The time budget that a server needs to handle client requests depends not only on the request rate, but also on the duration of each individual client request, which may itself depend on the input parameters. Therefore, our approach is to completely account the handling of a request to the initiating client. As a result, the worker threads in servers do not need a timeslice of their own, which reduces the number of timeslices in the system and simplifies their admission.

However, not every request in the system can be accounted to one particular client. For example, the handling of a disk interrupt that signals the completion of multiple read requests from different clients cannot be accounted to one particular client and would need to be split up among all involved clients. Other requests such as a network interrupt due to reception of unsolicited ARP traffic cannot be attributed to any client in the system. Because these events are inherently asynchronous and handled by the IRQ thread, we assign IRQ threads their own timeslice.

3. Kernel Requirements

In this section we deduce kernel requirements from the design decisions in the previous section. We show how the NOVA interface implements these requirements.

Multithreading

Because we build servers with multiple threads, the kernel should provide many lightweight threads. In order to handle independent client requests in parallel, a server could instantiate one worker thread per client. However, a large number of worker threads is often a waste of resources. Because each CPU can execute only one thread at a time, it is sufficient for servers to provide one worker thread per CPU.

The *helping* mechanism [10] in the kernel ensures that new client requests help to run the previous request on the CPU to completion. Helping works as follows: If a client wants to send a message to a worker that is currently busy handling another request, the client donates its timeslice to the preempted worker to help it finish the previous request. Once the worker becomes available again, it handles the request of the helping client. Helping makes accounting less precise because clients use their timeslice to help others. For clients with strict accounting requirements, the server can provide dedicated worker threads.

The NOVA microhypervisor supports multithreading by means of execution contexts. Each execution context can be created either with or without a timeslice. Execution contexts have a small memory footprint which can be as low as two memory pages — one for the thread control block and one for user-level data structures.

Synchronous/Asynchronous Communication

In a multi-server user-environment services are isolated from one another by means of address spaces. Communication between these services therefore requires a kernel mechanism for crossing address-space boundaries such as explicit IPC system calls, or shared memory and signaling. IPC is usually executed in two steps: the kernel first copies the message data and then transfers control from the sender to the receiver. The communication between two threads can be either synchronous or asynchronous.

In case of *synchronous* client-server communication, the client blocks after the data transfer and the kernel transfers control to the server. After sending the reply, the server blocks until the next request and the client regains control. If client and server are located on the same CPU, synchronous communication can be combined with timeslice donation, where the client donates its timeslice to the server for the duration of the request. If both threads are located on different CPUs, timeslice donation cannot be used. Furthermore, synchronous cross-CPU communication requires the kernel to perform an expensive remote unblock operation in both directions. If the communication between client and server is *asynchronous*, the client continues to execute after the message transfer. In that case the server needs its own timeslice to handle the request.

The NOVA interface supports synchronous IPC with timeslice donation between threads. It also provides asynchronous signaling via semaphores and mechanisms to establish shared memory between address-spaces.

Synchronization

When a server implements multiple threads on different processors, the kernel must provide an efficient synchronization mechanism that works across CPUs. NOVA supports cross-processor synchronization via semaphores. In a naive semaphore implementation, every `up()` or `down()` operation on a semaphore requires a system call. We implemented an optimized user semaphore that avoids the system call in the cases where `down()` only decrements the counter without blocking the caller, or where `up()` just increments the counter without releasing a thread blocked on the semaphore. The user semaphore consists of a kernel semaphore *ksem* and a user counter *ucount*, which resides in memory shared by all threads using the semaphore. If *ucount* is negative, it denotes the number of threads waiting to enter the critical section. Otherwise it indicates how many threads are still able to enter the critical section without blocking. The following code snippet shows the implementation of the user semaphore. The atomic *xadd* operation decrements or increments the counter and returns the previous value.

```
void usem::down() {
    if (atomic_xadd (&ucount, -1) <= 0)
        ksem->down();   // block me
}

void usem::up() {
    if (atomic_xadd (&ucount, +1) < 0)
        ksem->up();     // release someone
}
```

Because both functions can be preempted between the atomic counter update and the subsequent block/release operation or execute in parallel on a multiprocessor system, an interesting scenario can occur where a `down()`, which decrements *ucount* from 0 to -1 and then wants to block on *ksem*, races with an `up()`, which increments *ucount* from -1 to 0 and then wants to release a thread blocked on *ksem*. This race condition is automatically dealt with by the kernel semaphore that handles blocking and unblocking of threads. The interesting observation is that the order in which `ksem->down()` and `ksem->up()` execute does not matter, because they mutually neutralize their effects. Either `ksem->down()` puts the thread to sleep and `ksem->up()` will wake it up again, or `ksem->up()` increments the counter of the kernel semaphore and `ksem->down()` will decrement it again.

Our user-level semaphores are based on similar ideas as Linux Futexes [3]. The major difference is that in our implementation the kernel is completely unaware of the existence of the user counter and the associated user-level semaphore code.

4. Application

In the last section we described the low-level primitives that a kernel must provide to facilitate the construction of a scalable user-level environment for a multiprocessor platform. In this section we show how we use these primitives to construct services that run on top of the NOVA microhypervisor.

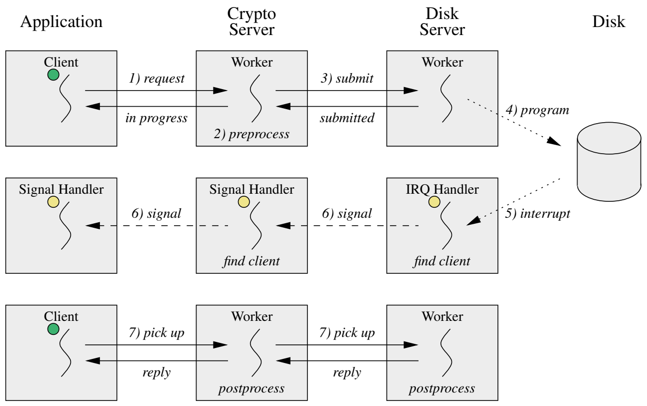

Figure 2: A client accesses a disk driver through a crypto server that implements transparent encryption and decrytion of disk blocks. An access consists of three phases: request, signal and pick up.

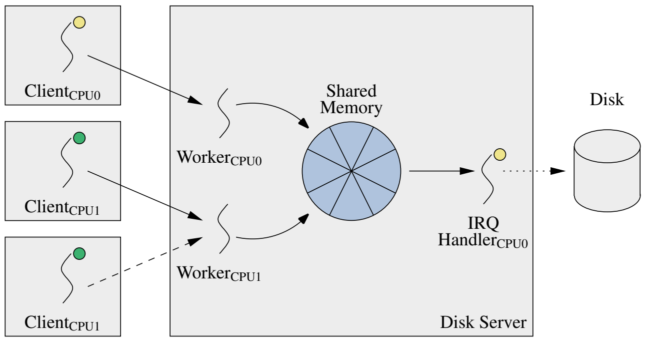

Figure 3: Request handling from different CPUs. One worker thread per CPU facilitates parallel processing of requests. The driver uses shared memory to internally forward the requests to the IRQ handler thread. The IRQ handler is the only thread that programs the hardware and synchronizes access to the device registers with the arrival of interrupts.

Client-Server Communication

Figure 2 depicts a client-server scenario in which a client wants to read a block from an encrypted disk. The encryption and decryption of disk blocks is handled transparently by an intermediate crypto server that is located between the client and the disk server that implements the device driver for the disk.

1) Request As a first step, the client sends a synchronous IPC request to the crypto server to request the block. Synchronous inter-process communication is similar to a remote procedure call. The client donates its timeslice to the worker thread in the crypto server and blocks until the crypto server responds.

2) Preprocessing After receiving the request, the worker thread in the crypto server can perform the necessary policy checks to determine if the client is allowed to access the disk block in question. If access is permitted, the worker can perform a buffer cache lookup to check if the block has been previously read from disk. If the block is available, the worker can directly respond to the client.

3) Submit Otherwise the crypto server sends a synchronous IPC to the disk server to read the block into memory. The timeslice that the worker received from the client is further donated to the disk server.

4) Program Upon receiving the request, the disk server programs the disk controller with commands to read the block. It should be noted that the entire handling of the disk request has been accounted to the timeslice of the client. After the disk server responds to the crypto server and the crypto server responds to the client, the client can continue executing its program while the disk is busy retrieving the block.

5) Interrupt Once the block has been read into memory, the disk controller generates an interrupt to signal completion of the request. The IRQ handler thread in the disk server is the handler for the disk interrupt. Because interrupts are asynchronous events, the IRQ handler thread has its own timeslice so that it can execute independently of any pending client requests.

6) Signal When one or more disk requests have been completed, the IRQ handler thread in the disk server sends a signal to the clients that submitted these requests. In our scenario the signal propagates via the signal handler thread in the crypto server to the signal handler in the client. Signal propagation uses semaphores instead of synchronous IPC because each signal handler may need to fan out the signal to multiple clients. Furthermore, servers do not necessarily trust their clients.

7) Pick Up After the client has received the signal that the read operation is complete, it can perform another synchronous IPC to the crypto server to retrieve the data. The purpose of the pick-up request is that the client is accounted for any postprocessing that is required during the transfer of the data from the disk server to the crypto server and back to the client. In our example, the decryption of the block is accounted to the client. It should be noted that the client can use a single IPC to batch multiple read-, write- and pick-up requests.

Internal Server Structure

Figure 3 shows the internal structure of a scalable disk driver that is accessible on CPU_0 and CPU_1. For this purpose the disk server creates a worker thread on each CPU. The worker threads listen for incoming requests from their respective CPU so that clients from different CPUs can contact the disk server concurrently. In our example, a client from CPU_0 and another client from CPU_1 simultaneously send a request to their corresponding worker thread in the disk server.

While handling the request of its client, the worker on CPU_1 is preempted by another client from CPU_1. The

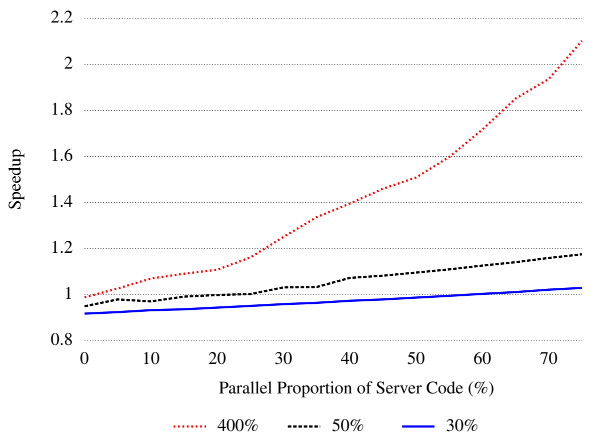

Figure 4: Speedup of a multi-threaded server over a single-threaded server for workloads where the server-internal cost is 400%, 50% and 30% of the cost of a cross-CPU call. The speedup depends on communication overhead and on how much of the server processing can be parallelized using per-CPU worker threads.

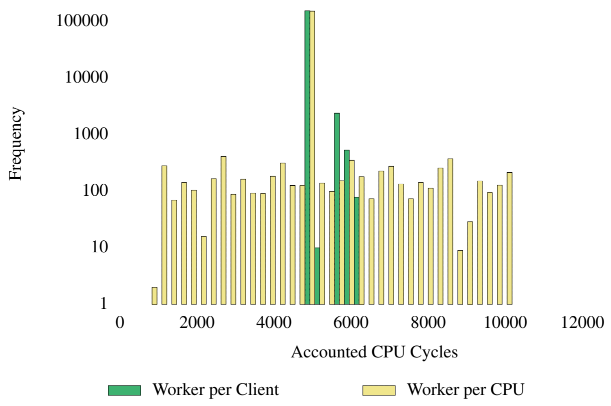

Figure 5: CPU cycles accounted to the client's timeslice for one server request of approximately 5000 cycles. With a worker thread per client, the client is accounted only for its own request. In servers with one worker per CPU a client may be accounted for up to twice as much if it needs to help the server finish the previous request.

second client uses its timeslice to help the previous client finish submitting its request (denoted by the dashed line).

All servers and device drivers share the common property that they provide one or more worker threads per CPU to handle concurrent client requests. However, the synchronization in each server is specific to the internal data structures and, in case of device drivers, the requirements of the hardware device. For example, a network driver that needs to program the same registers for receiving and transmitting packets synchronizes access to the device registers with a single handler thread. A driver for a more sophisticated network card with separate register files can use one thread for receiving and another thread for transmitting packets. Forwarding of requests from worker threads to handler threads uses shared memory and semaphores. The advantage of this approach is that the parallelism of client requests is preserved all the way to the device driver. The driver can then use its device-specific knowledge to serialize the requests and submit them to the hardware in the most efficient manner.

5. Evaluation

We evaluated an implementation of our proposed design using synthetic benchmarks on an Intel Core i7 system. Our test system had only 8 logical processors, but the scalability issues we discuss were already visible at this scale.

Single-Threaded vs. Multi-Threaded Server

In Figure 4 we compare the scalability of a single-threaded and a multi-threaded server. The single-threaded server handles all client requests serially with one thread on CPU_0. Clients located on CPU_1 through CPU_7 issue requests to the server using cross-processor IPC. For the multi-threaded server we added a worker thread on each CPU as illustrated in Figure 3. The worker threads accept client requests locally on their CPU, perform the parallel part of the processing and then forward the request cross-processor to the single thread on CPU_0, which handles the serial part of the processing. Compared to the single-threaded server, the multi-threaded server requires an additional IPC between the client and the worker thread, but can use the worker thread to parallelize a proportion of the server processing. Figure 4 illustrates that the single-threaded approach is only preferable for servers with a short critical section that cannot be parallelized. For example, if the duration of the server code path is 50% of the duration of a cross-processor IPC, then worker threads already improve the scalability if 20% or more of the server code can be executed in parallel. For longer server sections, more processors or a higher degree or parallelism in the server code, worker threads are always beneficial.

Worker Thread per Client vs. Worker Thread per CPU

If a server implements one worker thread per client, then clients only compete for CPU time, but not for server resources. If the server implements one worker thread per CPU, then a new client may need to help the worker thread finish the previous request on its CPU. Because all worker threads on the same CPU are subject to time-sharing, the server throughput is the same for both approaches. Figure 5 shows the distribution of accounted CPU cycles per client request. More than 99% of all client requests cost approximately 5000 cycles, which is the duration of one server request in our benchmark. Without helping there is a smaller peak at 6000 cycles, which includes the additional overhead of a timer interrupt that signals the end of timeslice.

The accounted time can be up to twice the duration of a server request in cases where the server is preempted immediately after accepting the previous request and the new client must help. The time accounted to a client may also be less than a server request when others help run the client's request to completion. In the case of helping, the distribution of accounted time is roughly uniform, because the server can be preempted anywhere in its code path.

6. Related Work

Existing work on designing efficient multi-processor operating systems has mostly focused on improving the scalability of the kernel. For example, K42 [7] achieves operating-system scalability through partitioning and replication of state at the kernel-object level. Corey [11] pushes the sharing policy to user level and allows an application to specify how OS resources are shared between different cores. Barrelfish [1] treats a platform with multiple heterogeneous processors as a distributed systems and replicates kernel state on every node. The nodes communicate with each other solely through message passing. Our work focuses on the design of user-level services, rather than kernel design.

We are currently not aware of any related work that explicitly targets multiserver environments on top of multi-core hardware. However, many systems move the traditional OS services to the application level and decompose them into multiple servers. Minix3 [6] uses the multiserver approach to increase the reliability of the system in the presence of faulty device drivers. SawMill [4] showed how the performance overhead of additional communication in a multiserver system can be reduced. The Bastei architecture [2] achieves a minimal application-specific trusted computing base with a multiserver design. Other research in DROPS [5] and Nemesis [8] focused on achieving quality-of-service guarantees for multimedia applications in a multiserver environment.

7. Conclusion

In this paper we outlined how a scalable user-level environment for a multiprocessor architecture can be built. By splitting servers into worker and IRQ handler threads, we can scale the performance of servers with the number of clients. Synchronous communication with timeslice donation facilitates precise accounting of compute-bound requests to the client that issued them. In our implementation, clients do not need to block on outstanding I/O. Instead, driver threads will signal the completion of requests asynchronously.

We discussed what mechanisms a kernel must provide in order to support the construction of a scalable user-level environment and how they can be applied in the context of the NOVA microhypervisor. We are currently applying the construction principles described in this paper in our user-level environment. In the future we plan to extend this work by offloading I/O to dedicated physical processors. Furthermore, we research how the timeslices of IRQ handler threads should be configured to guarantee certain quality-of-service constraints.

References

[1] 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. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[2] N. Feske and C. Helmuth. Design of the Bastei OS Architecture. Technical Report TUD–FI06–07, TU Dresden, 2006.

[3] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 479–495, 2002.

[4] 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 *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 109–114. ACM, 2000.

[5] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *ACM SIGOPS European Workshop*, pages 203–209, 1998.

[6] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proceedings of the 6th European Dependable Computing Conference (EDCC)*, pages 3–12. IEEE Computer Society, 2006.

[7] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. *SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[8] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[9] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010.

[10] U. Steinberg, J. Wolter, and H. Härtig. Fast Component Interaction for Real-Time Systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pages 89–97. IEEE Computer Society, 2005.

[11] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 43–57. USENIX Association, 2008.