Decoupled: Low-Effort Noise-Free Execution on Commodity Systems

Adam Lackorzynski, Carsten Weinhold, Hermann Härtig
Operating Systems Group, TU Dresden, Germany
{adam.lackorzynski,carsten.weinhold,hermann.haertig}@tu-dresden.de

ABSTRACT

Today's high-performance computing (HPC) landscape is dominated by clusters built from commodity hardware. The nodes of these systems are essentially x86-based servers that run an operating system (OS) derived from an enterprise Linux distribution. In contrast, previous generations of supercomputers ran OSes that were designed specifically for the needs of HPC applications. The migration from these special-purpose OSes to off-the-shelf system software brought many advantages for both vendors and users, most importantly reduced costs and a larger feature set. However, it also left behind an important property: jitter-free execution of parallel programs.

This jitter, often called OS noise, causes slowdowns for many important applications and is expected to become a major obstacle to exascale computing. Therefore, several OS research projects aim at building light-weight kernels that provide HPC applications with a noise-free execution environment. Linux runs next to these new kernels and provides functionality that they (intentionally) do not implement. However, building these new kernels and all the required support infrastructure requires considerable development and maintenance effort.

We argue that a noise-free HPC OS can be built upon existing components with much less effort. In this paper, we describe a node OS that combines an off-the-shelf microkernel with a virtualized Linux kernel that provides rich functionality, including device drivers. We extended these two building blocks with a simple mechanism to decouple program execution from noisy Linux. We evaluate our prototype on a recently installed InfiniBand cluster.

1. INTRODUCTION

The target of our work is performance isolation of applications running on large high-performance computing (HPC) systems. A significant number of these applications follow the bulk-synchronous programming model (BSP), which is characterized by alternating compute and communication phases. Delays of just one or few of the processes cause all others to idle in the communication phase, as this part of the program's execution acts as a barrier. Other types of HPC applications such as stencil codes suffer from imbalanced execution times in a similar way due to communication dependencies, which cause delays to propagate across processes.

ROSS '16, June 1, 2016, Kyoto, Japan

ISBN 978-1-4503-4387-9/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2931088.2931095

Avoiding OS Noise. One source of such delays on contemporary HPC systems is execution-time jitter or so-called *OS noise*. It has been shown that even small delays caused by OS noise (e.g., interrupt handling, page reclaim, monitoring daemons) can cause significant slowdown of applications as the number of compute cores increases [1, 2]. The straightforward way to avoid this problem is to use an OS that does not produce noise, like the system software based on the Compute Node Kernel (CNK) [3]. Unfortunately, such OSes lack much of the functionality that users and application developers demand. A full-fledged Linux OS does offer it, though. Therefore, many HPC system vendors modify the Linux kernel in an attempt to meet these two contradicting goals: noise-free execution and rich functionality. However, their success is either limited or involves significant development and engineering effort. The HPC OS projects mOS [4], FusedOS [5], Kitten/Hobbes [6], and Riken's Manycore OS [7] try an alternative approach by combining a light-weight kernel (LWK) and Linux. The idea is that the LWK implements performance and latency-critical system calls and ensures that application threads are never interrupted by background activity of the Linux part of the OS. System calls that cannot be handled by the LWK are forwarded to Linux, either based on proxies running on Linux [5–7], or by migrating processes between the LWK and Linux side [4].

Nevertheless, these solutions require significant development effort just to build the new kernels; these LWKs often have to include functionality that is not strictly performance critical (e.g., signal handling or ptrace support) [7]. Porting device drivers to the LWK, maintaining them, or writing new ones from scratch is a huge undertaking, too [6]. As a result, some projects do not yet support cross-node communication due to still incomplete I/O support [4].

The Case for Reuse and Modification. We argue that reusing and adapting building blocks that already exist requires much less effort in order to build a low-noise OS platform. In particular, we propose to combine the readily available *L4/Fiasco.OC* microkernel [8] and a slightly adapted version of the paravirtualized *L⁴Linux* [9] kernel into a node OS for HPC systems. Both the microkernel and L⁴Linux have been in active development for almost two

decades by now and are commercially supported.

The L4 family of microkernels is well-suited for use as a LWK. L4 provides only address spaces, threads, and a CPU scheduler as the basic primitives to implement processes; inter-process communication facilities and shared memory enable cooperation between threads. Complemented by a minimal set of low-level I/O support, these primitives suffice to implement all other parts of the OS, including device drivers and any management policies, as a set of services running in deprivileged user mode. By means of virtualization, the L^{4}Linux kernel running on top of this basic platform can be reused as a provider of communication drivers (e.g., for InfiniBand cards) and commodity infrastructure, ranging from the TCP/IP stack to an unmodified MPI library.

As the microkernel itself does not perform any background work, it cannot cause any noise.[1] To make noise-sensitive HPC applications running on L^{4}Linux benefit from this property, we propose the following rather small modifications:

1. The ability to *decouple* a thread from Linux' scheduling regime and move it onto a core that is under exclusive control of the microkernel.

2. A generic and light-weight way to *forward* system calls to L^{4}Linux whenever a decoupled thread needs to call the Linux kernel.

By decoupling all threads of, for example, an MPI application, most of the compute load on a node is taken away from the Linux environment. As a result, it is possible to restrict the number of cores that the L^{4}Linux scheduler sees and manages (e.g., just one or two), while decoupled threads run undisturbed on L4-controlled CPUs.

Contribution. The main contribution of this paper is an architecture for a node OS that efficiently supports noise-sensitive HPC applications on top of only slightly modified building blocks. We implement *decoupling* of threads from Linux on top of an existing microkernel. This approach is elegant and much simpler than building a new LWK; it allows us to reuse device drivers, provided by L^{4}Linux through paravirtualization, and runtime systems such as MPI with little or no modification. We evaluate our solution based on measurements on an HPC InfiniBand cluster.

Paper Structure. The remainder of this paper is structured as follows: In the next section, we give an overview of OS noise and the building blocks we use. Then we describe design and implementation of our decoupling approach in Section 3 and evaluate our prototype in Section 4. We discuss related work in Section 5 before we conclude and give an outlook on future work.

2. BACKGROUND

To help put our work into context, we discuss possible sources of system noise and give an overview of the L4 microkernel and L^{4}Linux.

2.1 Sources of Noise

Execution-time jitter in applications can be caused by both software and hardware. The former is called *OS noise* in the HPC community and includes maintenance inside the kernel such as page reclaim, internal synchronization, handling of interrupts (e.g., for periodic timers), and preemption to schedule system daemons. Firmware can also interrupt application threads, as code running in System Management Mode (SMM) on x86 platforms operates at higher privilege and priority levels than the OS kernel (i.e., even the OS can be interrupted). The particular choice of hardware platforms has influence on SMM-related noise. Fortunately, however, the most common reason of SMM activity is emulation of legacy input devices, which the OS can disable by initializing a USB driver.

Hardware-rooted disturbances include cache misses, cache evictions and bus congestion, also caused by activities on other cores. These causes are harder to avoid due to the close coupling of cores, sharing of caches, and memory controllers. Consequently, threads of the application may induce execution-time jitter due to their own activities. There are approaches for partitioning caches and memory, as well as techniques for multiplexing buses; however, they are out of scope for this paper.

In this paper, we address the issue of OS noise by restricting OS and firmware-related activities to those cores that do not execute noise-sensitive applications.

2.2 L4 Microkernel and L^{4}Linux

In this subsection we will focus on the key concepts of L4 and L^{4}Linux that are needed to understand how threads can be decoupled from L^{4}Linux. For a more complete description of L4 concepts, we refer to other publications [10, 11].

L4 Microkernel. The L4/Fiasco.OC microkernel [8] supervises the system. Similarly to the concept of processes in Linux, it provides *tasks* that enforce spatial isolation using address spaces. One or more *threads* can be assigned to a task to enable program execution within an address space. Threads can communicate with each other using synchronous inter-process communication (IPC), asynchronous signaling, and shared memory. The microkernel translates hardware interrupts (IRQs) into IPC messages that any thread can receive. A set of libraries and user-level components called *L4 Runtime Environment (L4Re)* provides additional infrastructure for memory management, loading applications, and other system services.

L4/Fiasco.OC supports symmetric multi-processors (SMPs) through a model where address spaces span all cores that execute threads of the corresponding task. Hence, there is a single instance of the microkernel that controls all cores of the machine. IPC and interrupt delivery is supported on the same core and across cores. The microkernel schedules threads locally on each core, but it does not migrate them to a different processor unless explicitly requested.

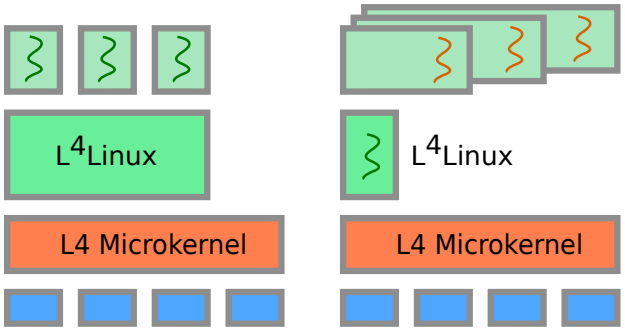
L^{4}Linux. Apart from supporting ordinary L4 applications, L4/Fiasco.OC can also act as a hypervisor. L^{4}Linux [9] is a para-virtualized Linux kernel that uses this capability; it has been modified to run as an L4 task on top of L4/Fiasco.OC and L4Re. L^{4}Linux supports unmodified Linux programs that execute in processes that are implemented as L4 tasks. All Linux processes are managed by L^{4}Linux, which loads the respective program binaries, maps and unmaps all memory pages, schedules their threads, and handles all Linux system calls.

From an architecture point of view, Linux processes are in the upper-most layer of the software stack on top of both

[1] By default, periodic timer interrupts occur on each core, but they can be disabled, if only one thread runs on the core.

L4/Fiasco.OC and the L^{4}Linux kernel; this is visualized in Figure 1(a). However, the para-virtualization approach makes it easy to move some of L^{4}Linux' responsibilities into the layer that is managed by L4. To decouple thread execution from the noisy L^{4}Linux environment, we can exploit the way in which the para-virtualized Linux handles executions of user-level programs: L^{4}Linux uses L4 threads in a special mode of operation, in which they behave like *virtual CPUs (vCPUs)* [12]. Each of these vCPU threads has associated with it a user-defined event handler for "exceptions"; L^{4}Linux registers with each vCPU a handler function for system calls and asynchronous events such as page faults or hardware interrupts. When L^{4}Linux is active and decides to schedule a thread of a user process, it migrates the currently active vCPU thread into the address space of that process, thereby leaving the L^{4}Linux kernel task in a way similar to executing the `sysexit` instruction on an x86 machine.

In effect, the infrastructure in L^{4}Linux for handling system calls and asynchronous events is largely transparent to L4/Fiasco.OC, as it uses basic L4 primitives. It can therefore be reused with little additional code from within L4 threads running directly on top of the microkernel.



(a) L^{4}Linux with three processes on vCPUs running on all cores (no decoupling)

(b) L^{4}Linux with three processes with decoupled threads on dedicated cores

Figure 1: Microkernel-based architecture with L^{4}Linux-based service OS (standard and decoupled)

3. DESIGN AND IMPLEMENTATION

We shall now give a brief overview of the software structure of nodes in an HPC cluster running our system. We then describe how we integrated our decoupling approach into it.

3.1 Architecture Overview

In our architecture, the L4 microkernel controls all cores of a node, whereas L^{4}Linux runs on very few, or potentially just one core. L^{4}Linux is the "service OS" that provides device drivers, network stacks, and application runtimes such as MPI. L4/Fiasco.OC isolates all components, including the L^{4}Linux kernel and HPC application processes, using address spaces and by enforcing thread placement. Therefore, spatial separation and performance isolation do not require static partitioning, but the assignment of cores can be changed during runtime by "hotplugging" vCPUs; soft-rebooting L^{4}Linux (but not the microkernel) is supported, too.

Interrupts from devices such as Ethernet adapters are routed to cores where L^{4}Linux is running; their IRQs do not influence the HPC applications running on compute cores. Device drivers for HPC network cards such as InfiniBand host channel adapters (HCA) require a different model. They typically consist of a Linux kernel driver and a user-level support library that is linked into the MPI library; both parts of the driver require direct access to the HCA in order to function and to achieve high performance. We therefore grant L^{4}Linux direct access to the InfiniBand card such that it can bind interrupts and access all I/O memory pages; the kernel driver then makes the I/O memory available to the user-level part of the driver via `mmap`. This scheme is used on native Linux to enable fast and low-latency communication, but it also works in L^{4}Linux[2] – even when processes linked against the user-space driver run decoupled from L^{4}Linux.

3.2 Decoupling Threads from L^{4}Linux

When we consider the L4 and L^{4}Linux fundamentals explained in Section 2.2, the way to decouple program execution of an already existing thread from L^{4}Linux' scheduler is simple: we must extend L^{4}Linux to create an alternative "execution context" for the noise-sensitive thread on a core that does not run any of its vCPU threads. To this end, our modified L^{4}Linux creates a new native L4 thread and inserts it into the address space of the user process. To ensure that at most one core executes code of the decoupled thread, we must prevent L^{4}Linux from scheduling this thread on its vCPUs. We achieve this by setting the state of the original thread context in L^{4}Linux to *uninterruptible*. After these two preparatory steps, our modified L^{4}Linux kernel migrates the newly created L4 thread to a dedicated core, where execution resumes. From then on both the new L4 thread and other L^{4}Linux threads can run in parallel; the decoupling operation is complete at this point. Figure 1(b) shows this for three processes with decoupled threads.

The decoupled thread can now run undisturbed until it triggers an exception, such as a system call, or when a page fault occurs. Note that accessing an already `mmap`'ed I/O memory page of, for example, the InfiniBand HCA does not trigger an exception; thus, the user-level driver linked into the HPC application can program directly through device registers the same I/O operations as on native Linux.

In principle, a decoupled thread can perform native L4 system calls, provided the necessary support is compiled into the program. However, the majority of system calls will actually be attempts to call the L^{4}Linux kernel, which L4/Fiasco.OC recognizes as "not an L4 system call"; it forwards any such exception to L^{4}Linux through an ordinary L4-IPC message. Upon receiving the exception, L^{4}Linux unblocks the process's original thread context, which it then schedules to perform the requested system call. Once the system call is completed, the Linux thread is marked uninterruptible again and the decoupled thread resumes execution after the user instruction that caused the exception.

3.3 Core-Local System-Call Handling

Linux already implements frequently-used system calls such as `gettimeofday` as user-level functions through a mechanism called *VDSO:* the Linux kernel maps executable code that implements such a system call into the address space of each

[2]We had to add 13 lines of code in the Mellanox InfiniBand mlx5 driver to establish additional in-kernel memory mappings to allow passing device memory to user-level programs on the para-virtualized L^{4}Linux kernel.

process. The VDSO mechanism is supported in L^{4}Linux and also works for decoupled threads that run on other cores; no inter-processor communication is required. Executing a non-VDSO system call in a decoupled thread does require cross-processor forwarding, which comes with an overhead over native Linux or L^{4}Linux. For time-critical or frequently-used operations, this overhead can be prohibitive. Thus, for applications that need fast handling of certain operations, it should be possible to execute the respective Linux system call on the core where the decoupled thread is running.

Currently, we do not support core-local execution of system calls. However, an implementation based on L4/Fiasco.OC and L^{4}Linux could work as follows: A second L4 thread is placed on the same core as the decoupled thread and configured as its exception handler; this handler thread is bound to the address space of the L^{4}Linux kernel and can thus access any of its data structures. All exceptions that are caused by the decoupled thread are forwarded to this handler thread, which inspects them. When local execution is possible, like in the case of `getpid`, the call is handled immediately by the exception-handler thread and no inter-processor communication takes place. However, care must be taken, as the exception handler thread is not running in a valid Linux context and thus cannot call arbitrary Linux APIs directly [13].

If the exception handler is unable to handle the request, it will be forwarded to the L^{4}Linux kernel. In this case, inspecting exceptions within the core-local handler thread increases the total time required to forward a system call by one kernel entry and exit. However, cross-core forwarding to a L^{4}Linux kernel thread still dominates system-call overhead, as it is an order of magnitude more expensive than local IPC in L4.

3.4 Implementation Details

In the following we will briefly describe interesting implementation details and further methods to reduce OS noise.

Signal Handling. Decoupled threads will not receive signals that have been posted to them until they enter the Linux kernel. Since they are not preempted, this might not happen at all. To force delivery of signals to decoupled threads, such as `SIGKILL`, we extended L^{4}Linux with a mechanism that scans for pending signals of decoupled threads. When detected, affected threads are forced to re-enter the L^{4}Linux scheduling domain, causing these signals to be handled.

Memory. All memory provided to Linux applications is managed by L^{4}Linux, even if threads are decoupled from its scheduler. As Linux may run page-replacement algorithms that cause additional page faults and thus disturbance, a noise-sensitive application may lock its virtual memory mappings using Linux's `mlock` and `mlockall` system calls. An application – or a memory-allocation library linked into it – may also request large pages in the same way as on native Linux in order to improve virtual memory performance.

Application Debugging. As the decoupling mechanism reflects any executed system call to the Linux kernel, any debugging utility can be used to debug decoupled applications. This is in contrast to other approaches (see Section 5) that need to explicitly implement debugging functionality.

3.5 User Interface

To support unmodified applications, we decided not to implement a new system call, but instead make the decoupling functionality accessible via the `sysfs` pseudo file system. The user can decouple a process's threads and move it to another host CPU as follows:

```
$ SYSFS_PATH=/sys/kernel/l4/decouple
$ echo $PID > $SYSFS_PATH/decouple
$ echo $CORE_ID > $SYSFS_PATH/$PID/cpu
```

An application that is already decoupled stays in this execution mode even after calling the `execve` system call. Thus, the user can create wrapper scripts to decouple applications without having to modify them:

```
#! /bin/sh
echo $$ > $SYSFS_PATH/decouple
echo $CORE_ID > $SYSFS_PATH/$$/cpu
exec "$@"
```

Multi-threaded applications require support for this interface in their thread-creation routines (e.g., libpthread) to place new threads on dedicated cores. Otherwise only the main thread will be decoupled; this was sufficient for our experiments.

4. EVALUATION

We evaluate our prototype by analyzing traces of execution-time jitter from both vendor-provided HPC Linux installations and our own system. We also study the impact of OS noise on application performance based on a simulated BSP workload. Furthermore, we quantify the overhead of system-call forwarding and discuss the complexity of the decoupling implementation.

4.1 Measuring OS Noise

To obtain high-resolution traces of execution-time jitter, we use the fixed-work quantum (FWQ) benchmark [14]. It measures repeatedly the time required to perform a fixed amount of work. On a noise-free system, each iteration of the work loop shall always take the same time. Any increase in execution time is caused by disturbances in the system (i.e., OS noise). We ran the benchmark on production systems of two HPC centers to understand how different hardware and software configurations influence OS noise. We calibrated the benchmark for the various hardware platforms such that one iteration of the loop (i.e., the "work") takes in the order of a millisecond; it executes the work quantum 10,000 times.

Taurus. We queued a single instance of FWQ into the batch system of the InfiniBand cluster *Taurus*, which is installed at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden. The nodes of this system have 2 Xeon® E5-2680 v3 processors with 12 cores each, 64 GiB RAM, and FDR Infiniband. The vendor-provided OS is based on RHEL 6.4 and a modified Linux 2.6.32 kernel. A trace of the measured OS noise is shown in Figure 2. The X axis indicates the iteration of the FWQ loop. The Y axis shows, compared to the minimal value that has been measured, the increase in run time in cycles (as reported by the `rdtsc` instruction) and the relative overhead in percent. In this setup, with the vendor-provided OS, system noise can increase the time to perform the work quantum by up to 8.7%, which translates to about 450,000 cycles or 200 μs.

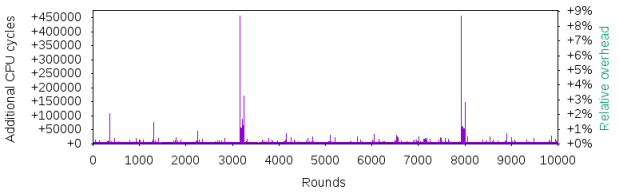


Figure 2: Noise pattern on Taurus, submitted through the batch system.

Workstation. We did initial tests of our decoupling implementation on a different system: a workstation with two Intel Xeon® X5650 6-core processors. When restricting L^{4}Linux to just one core, and with FWQ running decoupled on a core managed by the L4 microkernel, we observe significantly less noise than on the Taurus node. As shown in Figure 3, execution-time jitter drops to 30 to 50 cycles.

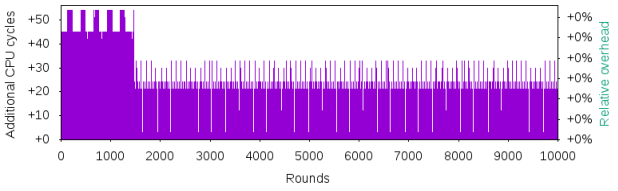


Figure 3: Same-socket noise pattern.

As shown in Figure 4, noise can be further reduced by placing the L^{4}Linux and the decoupled compute thread of FWQ on different sockets.

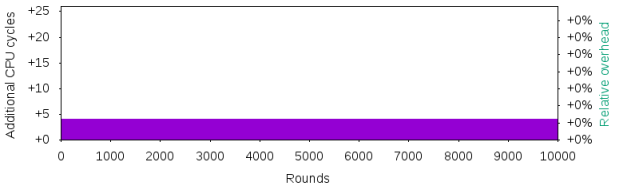


Figure 4: Different-socket noise pattern.

Apart from a 24 cycle spike in the first iteration of the FWQ loop that we believe to be a cache miss, the jitter never exceeds 4 cycles.

4.2 Noise Impact on BSP-Style Applications

The single-core tests described above show that there is potential to reduce noise on contemporary x86-based HPC systems. Therefore, we extended the FWQ benchmark into an MPI program called *MPI-FWQ*. This new benchmark executes its work loop in parallel on each MPI rank. It has two modes of operation that differ in how they use global MPI barriers; their behavior, which is visualized in Figure 5, is as follows:

- The *StartSync* mode uses a barrier across all MPI ranks once before entering the work.
- The *StepSync* mode additionally synchronizes all processes in a barrier before each iteration of the work loop.

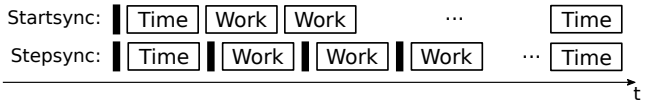


Figure 5: Behavior of MPI-FWQ in Startsync and Stepsync mode: each black bar depicts a global MPI barrier; timestamps are taken before the first and after the last iteration of the work loop.

In StartSync mode, MPI-FWQ simulates an embarrassingly-parallel application, whereas StepSync mode simulates a BSP-like application that alternates between computation and global synchronization.

Expectations. MPI-FWQ reports the minimum *(Min)* and maximum *(Max)* run times among all participating MPI processes. In StartSync mode, a difference between Min and Max indicates that there are cores in the system that are noisier than others; it shows the delta of aggregated noise impact between the "fastest" and the "slowest" cores. In contrast, when a loop iteration takes longer in StepSync mode for at least one MPI rank, all other ranks have to wait; any difference between Min and Max is then the delta for the last of the 10,000 loop iterations. We expect that all ranks run for approximately the same time in StepSync mode, but potentially longer than in StartSync due the additional barriers operations. The StepSync mode is particularly sensitive to OS noise as the number of compute cores increases.

Before evaluating our implementation of decoupling in L^{4}Linux, we first discuss the MPI-FWQ benchmark results for the vendor-provided OSes on HPC systems we had access to.

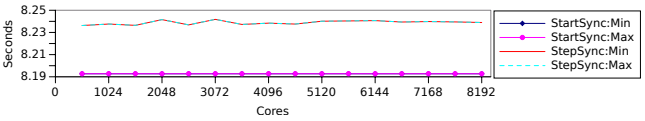


Figure 6: MPI-FWQ results for JUQUEEN.

Blue Gene/Q. The results for the Blue Gene/Q system *JUQUEEN* installed at the Julich Supercomputing Centre (JSC) are shown in Figure 6. The Y axis indicates the Min and Max values for both MPI-FWQ modes; the X axis indicates the number of cores used in each of the experiments (lines between measurement points are drawn for better readability). For Min and Max of the same mode, the results almost perfectly match, proving that JUQUEEN is indeed noise-free as we expected for this type of machine.[3] For reference, we present in Table 1 the *Min* and *Max* values and the delta between them; we only give the values for the experiment with the largest number of cores, which is 8,192.

X86-based Clusters. MPI-FWQ results for JSC's JURECA and ZIH's Taurus are shown in Figures 7 and 8, respectively. Both systems are x86-based InfiniBand clusters installed in 2015 and have similar hardware. Although their

[3]The PowerPC cores in the Blue Gene/Q system are significantly slower than than the x86 CPUs in machines we used for the single-core tests; due to a different calibration, the work quantum takes a shorter time to complete on JUQUEEN nevertheless.

	Min	Max	Delta
StartSync	8.192703s	8.192752s	0.000049s
StepSync	8.238959s	8.238969s	0.000010s
Delta (Mode)	0.046256s	0.046217s	

Table 1: JUQUEEN results for 8192 cores.

nodes use the exact same CPU model (2 Xeon® E5-2680 v3 12-core processors) and similar FDR InfiniBand cards, the run-times for our benchmark differ considerably. Also, JURECA shows a steep increase in run times as the number of involved cores increases. We did not investigate the reason for the deviations, but we suspect differences in the two systems' power-management configuration. They run different versions of the Linux kernel with different CPU drivers: JURECA's OS is based on Cent OS 7.2 and Linux 3.10, Taurus runs a 2.6.32-based Linux kernel.

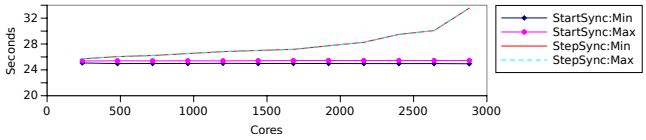


Figure 7: MPI-FWQ results for JURECA.

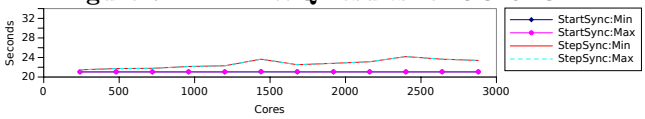


Figure 8: MPI-FWQ results for Taurus.

From the vastly different results for JURECA and Taurus, we conclude that we must evaluate our decoupling implementation in L^{4}Linux against a baseline that is as close to our prototype system as possible.

Custom Linux Distribution. We decided to use as baseline a custom Linux setup based on a recent L^{4}Linux 4.4 kernel for two reasons: First, there is no version of L^{4}Linux that is as old as the vendor kernels on Taurus or JURECA, but still compatible to the recent version of L4/Fiasco.OC. Second, we required a low-footprint Linux distribution that can be booted from the network. The reason for the second requirement is that we were lucky enough to get bare-metal access to nodes of ZIH's Taurus system, but installing a standard, multi-gigabyte Linux distribution on all the nodes was impracticable. Therefore, we stripped down a Debian software distribution such that it can completely operate in RAM. We compiled our own MPI library (MVAPICH 2.2b), which, in contrast to the one provided by the vendor of Taurus, is not tuned for this particular HPC system.

Bare-Metal Access. During the time window in which we had a chance to run our own OS on Taurus, we had access to an island of the cluster that had different nodes than the ones used for the benchmarks in Figure 8. These nodes are part of the phase-1 installation of Taurus and therefore have older CPUs, namely 2 Xeon® E5-2690 processors with 8 cores per socket; these CPUs operate at a higher clock rate than the E5-2680 v3 12-core processors.

Benchmark Configuration. Since we reserve one core for the noisy L^{4}Linux kernel, we could dedicate 15 cores of each node to MPI-FWQ; we ran the benchmark on up to 50 nodes. Despite the small number of nodes, we can report that decoupling the MPI processes of our benchmark does indeed improve performance over standard L^{4}Linux.

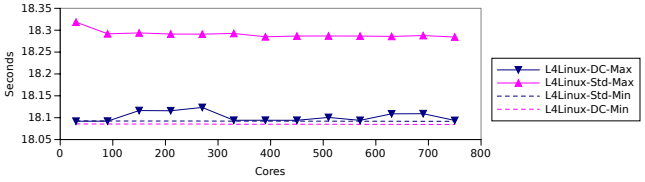


Figure 9: MPI-FWQ *StartSync* on standard (Std) and decoupled (DC) operation.

MPI-FWQ StartSync and Decoupled Execution. Figure 9 shows the results for the StartSync runs of MPI-FWQ for an increasing number of cores. The measurements labelled "Min" show that L^{4}Linux with decoupling enabled executes the MPI-FWQ benchmark slightly faster than standard L^{4}Linux. According to the Max values reported by MPI-FWQ, decoupled execution reduces noise significantly compared to the unmodified L^{4}Linux baseline: there is no "slow core" as we observe for the "L4Linux-Std-Max" run. Overall, MPI-FWQ shortens the maximum run time of the benchmark by 0.2 seconds or approximately 1 precent.

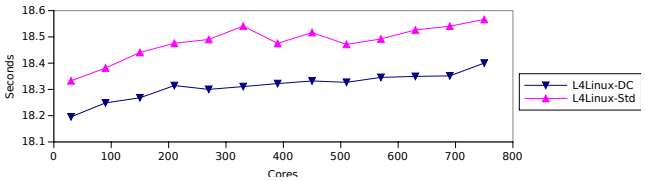


Figure 10: MPI-FWQ *StepSync* on standard (Std) and decoupled (DC) operation.

MPI-FWQ StepSync and Decoupled Execution. Figure 10 shows measured run times for the StepSync mode of MPI-FWQ. This workload simulates a bulk-synchronous application where all MPI processes synchronize at a global barrier almost every millisecond. For both standard L^{4}Linux and L^{4}Linux with decoupled execution, the Min and Max values are extremely similar; we therefore included in the diagram only the Min curves. Decoupled execution always comes out better in the benchmarks we were able to run while we had access to the Taurus nodes. The results confirm our expectation that the benchmark runs longer when the number of cores increases. By fitting a linear regression function into the delta between all data points in the graph, we can also conclude that the gap is widening (i.e., the OS-noise induced relative slowdown is reduced). This is in line with earlier research, for example by Seetharami et al. [1], which found that lower noise on each core leads to better performance. We expect to confirm the observed trend, when we repeat the experiment on a larger number of nodes the future.

We can conclude that our decoupling mechanism is indeed able to reduce OS noise and improve performance on x86-based HPC hardware. We therefore believe that the approach we presented in this paper is worthwhile and deserves further exploration, especially as its implementation involves only

small modifications to existing software compared to building a new LWK from scratch.

4.3 Cross-Core System Call Forwarding

When a program runs decoupled and issues a system call, the system call will take significantly longer to complete as it has to cross core boundaries twice. On an Intel Xeon X5650-based system in 64bit mode, we measure a system call latency for the `getpid` system call of about 14,600 CPU cycles for a same-socket placement and 21,500 cycles for placement on different sockets. Those results are in line with cross-core measurements of the basic L4 communication primitives which are dominated by Inter-Processor Interrupt (IPI) latency.

4.4 Development Effort

Due to the large reuse of existing code in L^{4}Linux, its prototypical status and some additional reorganization to add the decoupling mechanism, we can only give an approximate number of source code lines added. We believe the decoupling functionality required less than a thousand lines of code to be added to L^{4}Linux. Implementing the decoupling mechanism took a single, but knowledgeable developer about two weeks.

5. RELATED WORK

We already mentioned products such as IBM's Blue Gene supercomputers that physically separate the system into compute nodes and I/O nodes. Through this specific design, Blue Gene systems have very low OS noise. However, they also require specialized hardware and specialized system software, based on a LWK called Compute Node Kernel (CNK) [3], lacks convenience and features that developers know from Linux. In contrast, our approach uses off-the-shelf building blocks to provide HPC system software.

We currently see four operating system research projects targeting exascale computing, where avoiding OS noise is of great importance. Two of them, namely Argo and Hobbes, aim at optimizing resource usage through a high-level construct called enclaves, which is out of scope for this paper. Also, the Argo project [15] moved away from an LWK-based architecture in favor of cutting down Linux. The others build upon their own LWK and Linux:

Riken's Manycore OS. The Riken Exascale project includes both hardware and a systems software stack. The latter is based on Linux and a newly developed LWK called *McKernel* [16]. McKernel implements its own memory management, process and multi-threading support, a scheduler, and additional functionality, including POSIX signals and debug system calls such as `ptrace`. Forwarding of system calls that McKernel cannot handle is achieved by means of an Inter-Kernel Communication (IKC) layer; a proxy process on Linux is required for each process running on the LWK.

In contrast, our system runs the L4 microkernel on all cores and decoupled threads can move between L4 and Linux without the need for explicit forwarding or proxy processes.

Intel's mOS. mOS [4] shares with our system the approach to let processes migrate between the schedulers of Linux and the LWK. mOS achieves this by colocating a LWK with the Linux kernel and a system-call forwarding mechanism called *Evanescence*. An advantage of this colocation approach is that the LWK can directly access (and extend) data structures of the Linux kernel; for example, mOS controls thread placement using Linux' core-affinity mask.

The L^{4}Linux kernel and L4/Fiasco.OC are also tightly integrated through paravirtualization, which makes decoupling possible in the first place. However, to the best of our knowledge, mOS does not support yet running HPC applications across nodes due to incomplete I/O support.

Hobbes. Kitten [6] and Pisces are both built in the context of the Hobbes project [17]. Kitten is a LWK that has been designed to simplify the porting of Linux programs. Its interface is a small subset of POSIX tailored to provide operations critical to HPC codes; all other system calls are forwarded to Linux. Pisces supports running multiple kernels on virtual partitions of a node; it allows to allocate resources (cores and memory) to either Linux or Kitten.

In contrast, our architecture runs the L4 microkernel on all cores of a node. Separation between applications and L^{4}Linux is enforced through address spaces and thread placement.

Real Time. With performance isolation being the main objective in this paper, related work exists not only in the HPC world, but also in the area of real-time computing. For example, RTLinux [18] provides a real-time executive that runs the Linux kernel as a preemptible process side by side with hard real-time processes. However, from an architecture point of view, RTLinux is part of the Linux kernel. As a result, its real-time guarantees hold only as long as Linux continues to function and does not misbehave. DROPS [19] is more robust in that regard, as it relies on an earlier version of L4/Fiasco to enforce real-time properties. It combines a paravirtualized Linux with small real-time processes running on L4 directly. An important property of DROPS is its ability to support communication between real-time and non-real-time parts in a way that the real-time parts meet their deadlines under all possible behaviors of the non-real-time part, including crashes of Linux.

An import observation that lead to the DROPS architecture is that systems can often be split into two parts: a small part that is critical to enforce a specific property (e.g., real-time scheduling with L4) and a complex and functionally rich part that is not critical (e.g., a complete Linux kernel). In that sense, our decoupling approach is a successor to this earlier work on real-time computing, but with the concept of performance isolation now applied to HPC. That same idea of splitting systems into critical and non-critical components has also been applied to security [20].

6. CONCLUSIONS AND OUTLOOK

With our prototype we show that OS noise for HPC applications can be significantly reduced with low engineering effort by reusing existing OS technology. By enhancing virtualization mechanisms, we decouple Linux threads from the Linux kernel's scheduler and let them run undisturbed from OS noise caused by the commodity OS. All functionality offered by Linux is available to decoupled programs.

In future work, we will evaluate our prototype on significantly more nodes. We also plan to investigate how to extend hardware-assisted VM support for this use case by adopting mechanisms available for paravirtualized VMs. The system can be further optimized, for example, by reducing or avoiding the effect of the microkernel's timer interrupt. Using the system in other contexts, such as for real-time, looks interesting and promising as well.

7. ACKNOWLEDGMENTS

This research and the work presented in this paper is supported by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK [21]. We also thank the cluster of excellence "Center for Advancing Electronics Dresden" (*cfaed*). The authors acknowledge the Jülich Supercomputing Centre, the Gauss Centre for Supercomputing, and the John von Neumann Institute for Computing for providing compute time on the JUQUEEN and JURECA supercomputers. We would like to thank TU Dresden's ZIH deeply for giving us bare-metal access to nodes of their Taurus system, and Matthias Lange for creating our own Linux distribution.

8. REFERENCES

[1] Seetharami Seelam, Liana Fong, John Lewars, John Divirgilio, Brian F Veale, and Kevin Gildea. Characterization of System Services and Their Performance Impact in Multi-core Nodes. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 104–117. IEEE, 2011.

[2] Seetharami Seelam, Liana Fong, Asser Tantawi, John Lewars, John Divirgilio, and Kevin Gildea. Extreme scale computing: Modeling the impact of system noise in multi-core clustered systems. *Journal of Parallel and Distributed Computing*, 73(7):898–910, 2013.

[3] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10. IEEE, 2010.

[4] R.W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. mOS: An Architecture for Extreme-scale Operating Systems. In *Proc. ROSS '14*, pages 2:1–2:8. ACM, 2014.

[5] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W. Wisniewski. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 211–218, Washington, DC, USA, 2012. IEEE Computer Society.

[6] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S . Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS '10*, pages 1–12. IEEE, 2010.

[7] M. Sato, G. Fukazawa, K. Yoshinaga, Y. Tsujita, A. Hori, and M. Namiki. A hybrid operating system for a computing node with multi-core and many-core processors. In *Intl. J. Adv. Comput. Sci.*, volume 3, pages 368–377, 2013.

[8] Alexander Warg and Adam Lackorzynski. The Fiasco.OC Kernel and the L4 Runtime Environment (L4Re). avail. at https://l4re.org/.

[9] Adam Lackorzynski. L^{4}Linux. avail. at https://l4linux.org.

[10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[11] Adam Lackorzynski and Alexander Warg. Taming subsystems: Capabilities as Universal Resource Access Control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, Eurosys affiliated workshop*, IIES '09, pages 25–30. ACM, 3 2009.

[12] Adam Lackorzynski, Alexander Warg, and Michael Peter. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, 10 2010.

[13] Adam Lackorzynski. Managing Low Latency in Paravirtualized Virtual Machines. In *Proceedings of the 14th Real-Time Linux Workshop*, Chapell Hill, North Carolina, US, 2012.

[14] Lawrence Livermore National Laboratory. The FTQ/FWQ Benchmark.

[15] Pete Beckman et al. Argo: An Exascale Operating System and Runtime Research Project. Accessed 2016-03-07.

[16] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W. Wisniewski. Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[17] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt. Hobbes: Composition and Virtualization as the Foundations of an Extreme-scale OS/R. In *Proc. ROSS '13*, pages 2:1–2:8, 2013.

[18] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, 1 1997. The USENIX Association.

[19] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[20] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[21] FFMK Website. https://ffmk.tudos.org. Accessed 6 May 2016.