FFMK: A Fast and Fault-Tolerant Microkernel-Based System for Exascale Computing

Carsten Weinhold, Adam Lackorzynski, Jan Bierbaum, Martin Küttler, Maksym Planeta, Hannes Weisbach, Matthias Hille, Hermann Härtig, Alexander Margolin, Dror Sharf, Ely Levy, Pavel Gak, Amnon Barak, Masoud Gholami, Florian Schintke, Thorsten Schütt, Alexander Reinefeld, Matthias Lieber, and Wolfgang E. Nagel

Abstract The FFMK project designs, builds and evaluates a system-software architecture to address the challenges expected in Exascale systems. In particular, these challenges include performance losses caused by the much larger impact of runtime variability within applications, hardware, and operating system (OS), as well as increased vulnerability to failures. The FFMK OS platform is built upon a multi-kernel architecture, which combines the L4Re microkernel and a virtualized Linux kernel into a noise-free, yet feature-rich execution environment. It further includes global, distributed platform management and system-level optimization services that transparently minimize checkpoint/restart overhead for applications. The project also researched algorithms to make collective operations fault tolerant in presence of failing nodes. In this paper, we describe the basic components, algorithms, and services we developed in Phase 2 of the project.

1 Introduction

The operating system (OS) abstracts from low-level aspects of a computer system's hardware by providing applications with standardized programming interfaces and common services such as file systems and network access. By design, it

C. Weinhold (✉) · A. Lackorzynski · J. Bierbaum · M. Küttler · M. Planeta · H. Weisbach · M. Hille · H. Härtig · M. Lieber · W. E. Nagel
TU Dresden, Dresden, Germany
e-mail: carsten.weinhold@tu-dresden.de

A. Margolin · D. Sharf · E. Levy · P. Gak · A. Barak
The Hebrew University of Jerusalem, Jerusalem, Israel

M. Gholami · F. Schintke · T. Schütt · A. Reinefeld
Zuse Institute Berlin, Berlin, Germany

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing - SPPEXA 2016–2019*, Lecture Notes in Computational Science and Engineering 136,
https://doi.org/10.1007/978-3-030-47956-5_16

stands between the hardware and all applications. In the high-performance computing (HPC) community, the OS is therefore sometimes considered to be "in the way" as applications try to extract maximum performance from the underlying hardware. Indeed, the OS can introduce overhead, as we will discuss in the following. But challenges posed by upcoming Exascale systems such as load imbalances or failures due to increasing component counts can benefit from system-level support. Therefore, the central goal of the FFMK project has been to investigate how the OS can actually help, rather than be a source of overhead.

In the following paragraphs, we summarize the general architecture of the FFMK OS platform and give an overview of its higher-level services. In part, this description is an overview of results from Phase 1 of the project; but it shall also help put the results presented in this paper into context. For a much more detailed discussion of FFMK and the motivation behind it, we refer to our previous project report [59].

Multi-Kernel Node OS Figure 1 shows the architecture of the FFMK node OS. It is built on a multi-kernel foundation comprising an L4 microkernel and a variant of the Linux kernel that is called L^{4}Linux. We aim to support unmodified HPC applications and they shall have access to the same runtime libraries and communication drivers that are used on standard Linux-based HPC OSes. We target noise-sensitive applications by providing jitter-free execution directly on top our microkernel [33]. In this context, we also investigated the influence of hardware performance variation [61]. However, our vision for an HPC OS platform also includes new platform management services to support more complex and

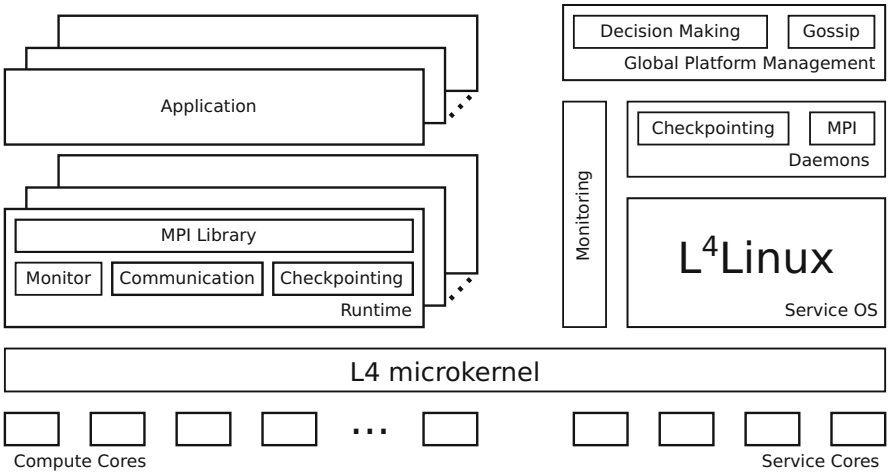

Fig. 1 FFMK software architecture: compute processes with performance-critical parts of (MPI) runtime and communication driver execute directly on L4 microkernel; functionality that is not critical for performance is offloaded to the L^{4}Linux kernel, which also hosts global platform management and fault-tolerance services

dynamic applications, as well as algorithms and system-level support to address fault-tolerance challenges posed by Exascale systems with unprecedented hardware component counts.

Applications, Runtimes, Communication HPC applications are highly specialized, but they achieve a certain level of platform independence by using common runtime and communication APIs such as the Message Passing Interface (MPI) [18]. However, just proving an MPI library and interconnect drivers (e.g., for InfiniBand) is not sufficient [60], because the majority of HPC codes use many Linux-specific APIs, too. The same is true for most HPC infrastructure, including parallel file systems and cluster management solutions. Compatibility to Linux is therefore essential and this can only be achieved, if applications are in fact compiled for Linux and started as Linux processes.

Dynamic Platform Management The FFMK OS platform is more than a multi-kernel architecture. As motivated in the Phase 1 report [59], we include distributed global management, because the system software is best suited to monitor health and load of nodes. This is in contrast to the current way of operating HPC clusters and supercomputers, where load balancing problems and fault tolerance are tasks that practically *every* application deals with on its own. In the presence of frequent component failures, hardware heterogeneity, and dynamic resource demands, applications can no longer assume that compute resources are assigned statically.

Load Balancing We aim to shift more coordination and decision making into the system layer. In the FFMK OS, the necessary monitoring and decision making is done at three levels: (1) on each node, (2) per application instance across multiple nodes, and (3) based on a global view by redundant master management nodes. We published fault-tolerant gossip algorithms [3] suitable for inter-node information dissemination and found that they have negligible performance overhead [36]. We further achieved promising results with regard to oversubscribing of cores, which can improve throughput for some applications [62]. We have since integrated the gossip algorithm, a per-node monitoring daemon, and a distributed decision making algorithm aimed at automatic, process-level load balancing for oversubscribed nodes. However, one key component of this platform management service is still missing: the ability to migrate processes from overloaded nodes to ones that have spare CPU cycles. Transparent migration of MPI processes that directly access InfiniBand hardware has proven to be extremely difficult. We leave this aspect for a future publication, but do we do summarize our key results on novel diffusion-based load balancing algorithms [39] in this report. These algorithms could be integrated into the FFMK load management service once process-level migration is possible.

Fault Tolerance The ability to migrate processes away from failing nodes can also be used for proactive fault tolerance. However, the focus of our research on system-level fault tolerance has been in two other areas. First, we published on efficient collective operations in the presence of failures [27, 31, 43]. Second, we continued research on scalable checkpointing, where we concentrated on global coordination

for user-level checkpointing [22] and how to optimize it based on expected failure rates [21].

In the following two sections of this paper, we discuss how re-architecting the OS kernel for HPC systems can improve performance and scalability (Sect. 2); we also present results on how to increase kernel scalability beyond the dozens of cores found in contemporary systems, as well as new load balancing algorithms. We then make the case that system-software support is essential to address fault-tolerance challenges posed by Exascale systems and how new fault-tolerant algorithms can help improve robustness and performance of collective operations (Sect. 3). Each individual subsection on these pages summarizes our peer-reviewed work in the respective area.

2 Building Blocks for a Scalable HPC Operating System

2.1 The Case for a Multi-Kernel Operating System

Noise-Sensitive Applications A widely reported problem in HPC is "OS noise" [5, 16, 26, 49, 53], where sporadic or periodic housekeeping activities of the OS (or other background tasks) briefly interrupt application threads. These interruptions can slow down applications based on the bulk-synchronous programming (BSP) model. BSP applications are parallel programs that are characterized by alternating computation and communication phases that all participating threads must perform in perfect synchronization to maximize throughput. If just a few compute threads are preempted by background activities, all other threads that depend on their input will waste CPU cycles as they wait for the stragglers to reach the communication phase. As shown in Fig. 2, delays can amount to hundreds of thousands of cycles, resulting in a slowdown of up to 9% for a computation that takes 1.5 ms to complete when there is no interruption.

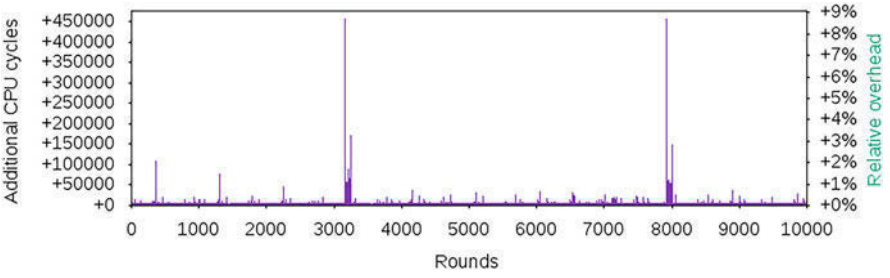

Fig. 2 OS noise during a run of the fixed work quantum (FWQ) benchmark on a node of a production HPC cluster with Linux-based vendor OS

Re-Architecting the Kernel for HPC One way to address the OS noise problem is to partition the compute resources of each node into two sets of cores: *compute cores* that are allocated exclusively to HPC applications and *service cores* that run management and maintenance tasks. Multi-kernel OS architectures implement this approach by assigning the compute cores to a lightweight kernel (LWK); a traditional kernel such as Linux runs management and node monitoring daemons on the service cores. The LWK does not preempt compute threads, thereby minimizing execution-time jitter for applications. However, a LWK for HPC does not replace a complex kernel such as Linux. It only implements functionality that is critical for application performance; system calls that do not impact performance are offloaded to the Linux kernel running on the service cores. The multi-kernel approach gives HPC applications the best of both worlds: the LWK ensures low noise and high performance, whereas Linux offers convenience, familiar APIs, a rich feature set, and compatibility with huge amounts of legacy infrastructure.

A Microkernel as a Universal LWK Multi-kernel OS architectures have received a lot of attention in recent years and several new LWKs have been developed, including IHK/McKernel [55] and mOS [63]. However, the maintenance effort for keeping an HPC-suitable multi-kernel OS up to date and compatible with Linux must be smaller than constantly patching mainline Linux to make it less noisy. Arguably, the best way to meet this requirement is to reuse components that already exist and that are actively maintained. In the FFMK project, we therefore chose to use the mature L4Re microkernel and L^{4}Linux as the basis of the FFMK node OS. In contrast to McKernel and mOS, the L4Re microkernel manages not just the designated compute cores, but all processor resources. The L^{4}Linux kernel runs virtualized on top of L4Re as shown in Fig. 1 on page 484.

2.2 L4Re Microkernel and L^{4}Linux

In this Subsection, we quote[1] from a previous publication [60] an overview of the L4Re ecosystem, including the L4Re microkernel and the paravirtualized Linux kernel L^{4}Linux. These two basic building blocks are combined into a foundation of a highly flexible and low-noise node OS. In Sect. 2.3, we evaluate the main benefits of this multi-kernel architecture.

L4 Microkernel The L4Re microkernel is a member of the L4 family of microkernels. The core principle of L4 [40] is that the kernel should provide only the minimal amount of functionality that is necessary to build a complete OS on top of it. Thus, an L4 microkernel is not intended to be a minimized Unix, but instead it provides only a few basic abstractions: address spaces, threads, and inter-

[1]This description has been shortened and slightly edited for brevity; see [60] for the complete version.

process communication (IPC). For performance reasons, a thread scheduler is also implemented within the kernel. However, other OS functionality such as device drivers, memory management, or file systems are provided by system services running as user-level programs on top of the microkernel.

Applications and User-Level Services L4Re applications communicate with each other and with system services by exchanging IPC messages. These IPC messages can not only carry ordinary data, but they may also transfer access rights for resources. Being able to map memory pages via IPC allows any two programs to establish shared memory between their address spaces. Furthermore, because it is possible to revoke memory mappings at any time, this features enables user-level services to implement arbitrary memory-management policies. In much the same way an L4Re program can pass a *capability* referencing a resource to another application or service, thereby granting the receiver the permission to access that resource. A capability can refer to a kernel object such as a `Thread` or a `Task`, representing an independent flow of execution or an address space, respectively. But they may also point to an `Ipc_gate`, which is a communication endpoint through which any user-space program can offer an arbitrary service to whomever possesses the corresponding capability.

I/O Device Support An important feature of the L4Re microkernel is that it maps hardware interrupts to IPC messages. A thread running in user space can receive interrupts by waiting for messages from an `Irq` kernel object. In conjunction with the possibility to map I/O memory regions of hardware devices directly into user address spaces, it is possible to implement device drivers outside the microkernel.

Virtualized Linux The L4Re microkernel is a fully functional hypervisor capable of hosting virtual machines running unmodified guest operating systems. It employs hardware-assisted virtualization on instruction set architectures that support it, including x86, ARM, and MIPS. Device emulation or passthrough is supported through virtual machine monitors running in user space. However, faithful virtualization is not the only way to run a legacy OS on top of the L4Re microkernel. L^4Linux is a paravirtualized Linux kernel that has been adapted to run on the interfaces provided by L4Re. It is binary compatible with standard Linux programs, however, instead of running in the privileged mode of the CPU, the L^4Linux kernel runs as a multi-threaded user-level program. Linux user processes run in their own L4 tasks (i.e., other address spaces). Linux programs on L^4Linux experience the same protection as on native Linux; they cannot read or write the Linux kernel's memory and they are protected from each other like processes on native Linux.

The vCPU Mechanism For execution, L^4Linux employs vCPUs, a mechanism provided by the microkernel that allows for an asynchronous execution model where a vCPU migrates between executing code in the L^4Linux kernel and user code in Linux processes. For any event that needs to be handled by the Linux kernel, such as system calls and page faults by processes, or external interrupts by devices, the vCPU switches to the L^4Linux kernel to handle them.

L^{4}Linux Process Model L^{4}Linux manages the address spaces of Linux user processes through `Task` objects provided by the L4Re microkernel. Thus, every Linux process and the contents of its address space are known to the microkernel. Furthermore, L^{4}Linux multiplexes all user-level threads executing in such an address space onto its vCPUs. Thus, the L4Re microkernel is involved in every context switch of any Linux user thread. In particular, it is responsible for forwarding any exceptions raised by a Linux program to the L^{4}Linux kernel. Exceptions occur when a thread makes a system call, when a page fault occurs during its execution, or when a hardware device signals an interrupt. L^{4}Linux receives these exceptions at a previously registered vCPU entry point, to which the microkernel switches the control flow when it migrates the vCPU from the `Task` of the faulting Linux user program to the address space of the virtualized Linux kernel.

2.3 Decoupled Execution on L^{4}Linux

Since virtually all HPC codes are developed for Linux and require so many of its APIs, the only practical option is to execute them on a Linux-based OS. Just running them on L^{4}Linux yields no benefit. However, the tight interaction between the L4Re microkernel and L^{4}Linux allows us to conveniently implement a new mechanism we call *decoupling* [33].

Decoupling Thread Execution from L^{4}Linux The purpose of decoupling is to separate execution of a thread in a Linux process from the vCPU it is normally running on. To this end, we create a separate, native L4Re host thread that runs in the same L4 `Task` (i.e., address space) as the Linux process, but not under control of L^{4}Linux. The original Linux thread context in the L^{4}Linux kernel is suspended while execution progresses on the native L4Re host thread. The user code running there will raise exceptions just as if it were executed by a vCPU, except that the microkernel forwards each of them to L^{4}Linux as an *exception IPC message*. A message of this type carries a thread's register state and fault information as its payload, and is delivered by the microkernel to an exception handler. We configure L^{4}Linux to be the exception handler of the "decoupled" Linux user threads. An attempt to perform a Linux system call will also result in an exception, which the L^{4}Linux kernel can then handle by briefly reactivating the previously suspended thread context and scheduling it onto a vCPU. Afterwards, execution is resumed in decoupled mode on the L4Re host thread. Figure 3 visualizes decoupled execution; more details can be found our publications [33, 60].

One Mechanism for the Best of Both Worlds The net gain of the decoupling mechanism is that we can combine noise-free execution on our LWK (i.e., the L4Re microkernel) with the rich execution environment of Linux, including all its APIs and the HPC infrastructure built for it. Decoupled threads use a *single* mechanism for forwarding any system call or exception, instead of many specialized

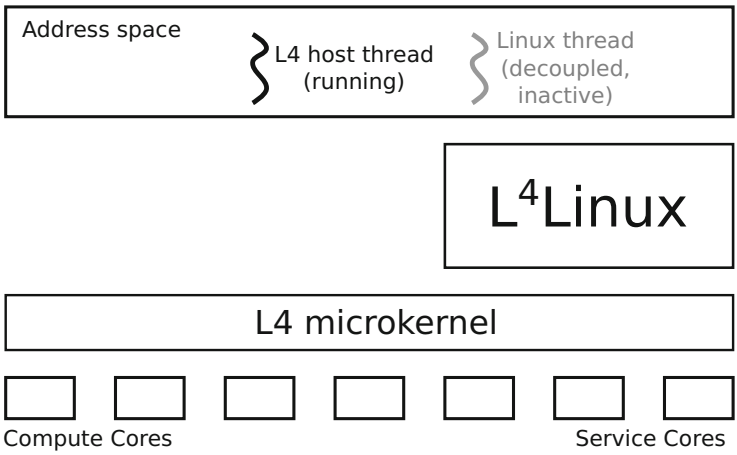

Fig. 3 Schematic view of the decoupling mechanism. The L4Re microkernel runs on every core of the system, while the virtualized L4Linux runs on a subset of those cores only. All normal Linux applications are thus restricted to those cores. Decoupling pulls off threads and runs them on cores not available to L4Linux

proxies that other multi-kernel HPC OSes use and that are difficult to maintain [60]. Applications are built for Linux and start running as Linux processes, but we pull their threads out of Linux's scheduling regime so they can run on dedicated cores without being disturbed by L4Linux. Effectively, decoupled threads run directly on the microkernel. However, they can use all services provided by L4Linux, which will continue to handle Linux system calls and resolve page faults. Also, since the InfiniBand driver in the L4Linux kernel maps the I/O registers of the HCA into the address space of each MPI rank, the high performance and minimal latency of the user-space part of the driver is not impaired; a decoupled thread can program performance-critical operations just like it would on native Linux.

CPU Resource Control The number of vCPUs assigned to L4Linux and their pinning to physical CPU cores determines how much hardware parallelism an L4Linux-based virtual machine can use. All other cores are exclusively under control of the L4Re microkernel and can therefore be allocated exclusively to decoupled threads of HPC application processes.

Initial Benchmark We used the fixed-work quantum (FWQ) [35] benchmark to determine how much "OS noise" decoupled threads experience. FWQ executes a fixed amount of work in a loop, which on a perfectly noise-free system should require a constant amount of CPU cycles to complete. Figure 4 shows the result of our first benchmark run, performed on a 2-socket machine from our lab. Using the `rdtsc` instruction, we measured delays of up to 55 CPU cycles per iteration when FWQ is executed by a decoupled thread on a dedicated core. The execution-time jitter is reduced to 4 cycles per iteration, when FWQ is offloaded to the second socket, while L4Linux is pinned to a single core of socket 1.

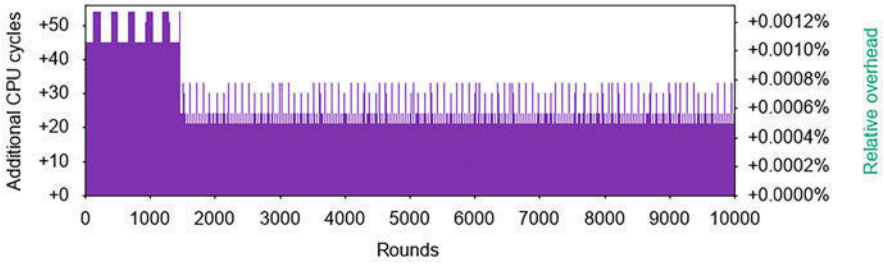

Fig. 4 Minimal OS noise remaining in decoupled execution; L^{4}Linux running on same socket

Multi-Node Benchmark The execution-time jitter we measured in the lab is four to fives orders of magnitude smaller than what we saw when FWQ ran on the vendor-provided Linux OS of Taurus, an HPC cluster installed at TU Dresden (see Fig. 2 on page 486). For larger-scale benchmarks, we extended FWQ into *MPI-FWQ*, a parallel benchmark that executes work-loop iterations in each MPI process on all participating cores. We performed experiments with our L4Re/L^{4}Linux-based node OS on 50 Taurus nodes. Each node has two Xeon® E5-2690 processors with 8 cores per socket. To benchmark "decoupling" in a parallel application, we allocated one core to L^{4}Linux and the remaining 15 cores to MPI-FWQ. The baseline we compare against is 15 MPI-FWQ processes per node scheduled by the same L^{4}Linux on the same hardware, but in a 16-vCPU configuration with no decoupled threads.

EP and BSP Runs MPI-FWQ can operate in two modes: In *StartSync* mode, a single barrier across all ranks is used to synchronize once when the benchmark starts; this mode simulates an embarrassingly parallel (EP) application. In *StepSync* mode, MPI-FWQ waits on a barrier after each iteration of the work loop, thereby acting like an application based on the bulk-synchronous programming (BSP) model.

Figure 5 shows the time to completion for any MPI-FWQ process operating in BSP-style StepSync mode, as we increased the total number of MPI processes (i.e., cores) from 30 to 750. The benchmark run time with decoupled MPI-FWQ threads (L4Linux-DC) is approximately 1% shorter than when the standard scheduler in the L^{4}Linux kernel controlled application threads (L4Linux-Std); results for EP-style StartSync runs show a similar performance. This difference is smaller than the up to 9% jitter we saw with the vendor OS, but our stripped down Linux environment lacks most of the management services and system daemons that run on the cluster. These services could never preempt *decoupled* threads, though.

More information on the decoupling mechanism, additional use-cases, and evaluations results can be found in separate publications [32–34].

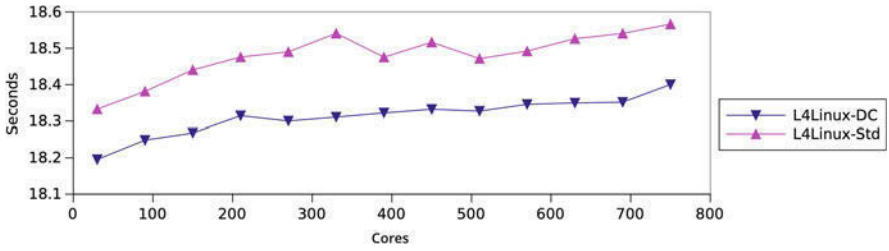

Fig. 5 BSP-style MPI-FWQ (*StepSync* mode) on L⁴Linux (Std) and with decoupled thread execution (DC) on Taurus. This figure has originally been published in [33]

2.4 Hardware Performance Variation

Software-induced imbalance in large-scale parallel simulations have also been studied by other groups, who proposed multi-kernel architectures as well [20, 23, 30, 48, 50, 51]. However, one source of variability has not been systematically investigated so far: the hardware itself. Hardware performance variation is particularly interesting due to growing diversity of platforms in HPC and the increasing complexity of computer architectures in general.

Measuring Hardware Performance Variation Characterizing hardware performance variability is challenging, because it requires a tightly controlled software environment. LWKs [51] have the greatest potential to obtain a precise characterization of various aspects of hardware performance variability on real HPC hardware. Towards this end, we have developed an extensible benchmarking framework to systematically characterize different aspects of hardware performance variability [61]. We use this benchmark suite to analyze five platforms described in Table 1: Intel Xeon [29], Intel Xeon Phi [56], Cavium ThunderX [9], Fujitsu FX100 [64] and IBM BlueGene/Q [28]. To minimize "OS noise", we benchmarked on OSes based on two LWKs: CNK on the IBM BG/Q and IHK/McKernel [20, 55] on the Intel, Fujitsu, and Cavium machines.

Benchmark Suite In addition to the previously described FWQ benchmark, our benchmark suite consists of seven other benchmark kernels. They were selected from well-known algorithms, micro benchmarks, or proxy applications and have the following characteristics:

- The **DGEMM** benchmark performs matrix multiplication. We confine ourselves to naïve matrix multiplication algorithms to allow compilers to emit SIMD instructions, if possible. This benchmark kernel is intended to measure hardware performance variation for double-precision floating point and vector operations. The **SHA** algorithm utilizes integer execution units instead.

Table 1 Summary of architectures

Platform/property	Intel Ivy Bridge	Intel KNL	Fujitsu FX100	Cavium ThunderX	IBM BG/Q
ISA	×86	×86	SPARC	ARM	PowerISA
Number of cores	8	64 + 4	32 + 2	48	16 + 2
Number of SMT threads	2	4	N/A	N/A	4
Clock frequency	2.6 GHz	1.4 GHz	2.2 GHz	2.0 GHz	1.6 GHz
L1d size	32 kB	32 kB	64 kB	32 kB	16 kB
L1i size	32 kB	32 kB	64 kB	78 kB	16 kB
L2 size	256 kB	1 MB × 34	24 MB	16 MB	32 MB
L3 size	20480 kB	N/A	N/A	N/A	N/A
On-chip network	Ring	2D mesh	unkown	Ring	Cross-bar
Process technology	22 nm	14 nm	20 nm	28 nm	45 nm

- Using John McCalpin's **STREAM** benchmark, we assess variability in the cache and memory subsystems. The **Capacity** benchmark is intended to measure performance variation of cache misses themselves.
- **HACCmk** from the CORAL benchmark suite is a compute-intensive N-body simulation kernel with regular memory accesses. **HPCCG** from Mantevo's benchmark suite is a Mini-App aimed at exhibiting the performance properties of real-world physics codes working on unstructured grid problems. **MiniFE** is another proxy application for unstructured implicit finite-element codes from Mantevo's package.

More detailed descriptions of these benchmark kernels, modifications we made, and all measurement results can be found in our paper [61]. In this report, we highlight only a few key results from FWQ, HPCCG, and DGEMM experiments.

Workload Matters Like previous studies on software-induced performance variation, we relied on the FWQ benchmark to evaluate execution-time jitter for decoupled threads. However, this simple benchmark kernel, may be suitable to quantify interruptions caused by system software, but they are insufficient to capture hardware-induced noise. We hypothesize that the full extent of hardware performance variation can only be observed when the resources which cause these variations are actually used. And indeed, as shown in Fig. 6, the HPCCG proxy code running on IHK/McKernel on an Intel Ivy Bridge system shows about 1% of performance variation among all cores of a node.

We measured variation on 30 SMT cores of this 2-socket Intel Ivy Bridge E5-2650 v2 system for both FWQ and HPCCG. We set the working set size of HPCCG to 70% of the L1 data cache size (32 KiB), disabled TurboBoost, set the scaling governor to *performance*, and fixed the clock speed to the nominal frequency of 2.6 GHz. We additionally sampled the performance counters for L1 data cache and L1 instruction cache misses and confirmed that both benchmarks experience little to no misses, in particular cores one to seven and 16 to 29 experience neither instruction cache nor data cache misses under HPCCG. Nevertheless all cores show significantly more variation under HPCCG than under FWQ. We conclude that FWQ is indeed ill-suited to measure hardware-induced performance variation.

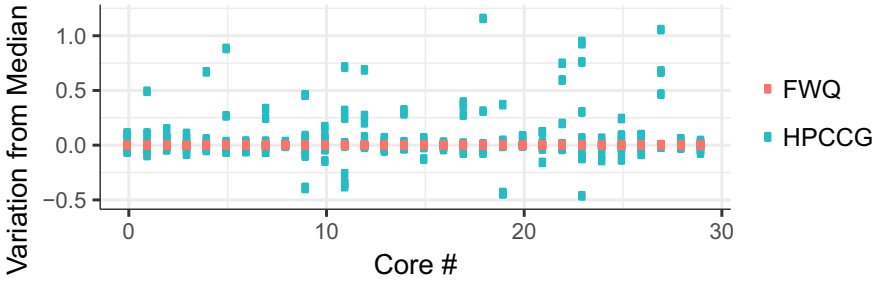

Fig. 6 Performance variation of FWQ and HPCCG on a dual-socket Intel E5-2650 v2

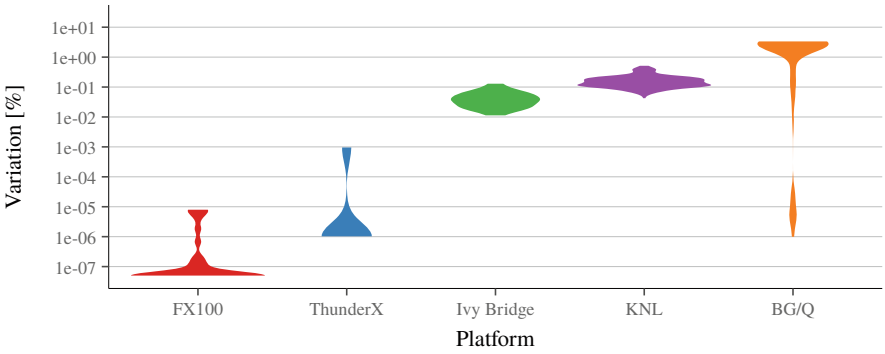

Fig. 7 Hardware performance variation under the DGEMM benchmark

Microarchitectures Differ Figure 7 visualizes our measurements for the DGEMM benchmark, which is dominated by floating point operations. We observe that the FX and ThunderX platforms exhibit very low variation, and note the rather high variation of the Ivy Bridge, KNL and BlueGene/Q platforms. We saw high numbers of cache misses on the Ivy Bridge platforms and therefore reduced the cache pressure to 70% fill level. After this modification to the benchmark setup, we saw stable or even zero cache miss numbers for all cores of the Ivy Bridge platform, but variation did not improve. We conclude that the measured variation is not caused by the memory subsystem.

Overall we found that just focusing on CPU core-local resources, like we did in this study, already shows up to six orders of magnitude difference in relative variation among different CPU architectures.

2.5 Scalable Diffusion-Based Load Balancing

A low-noise execution environment is essential for certain types of applications, as we explained in the preceding subsections. However, there are other HPC codes where noise is less of a concern, because they suffer from load imbalances that are inherent to the problem domain. For example, they might use multiple simulation kernels with different computational complexity. Furthermore, some computations are difficult to parallelize efficiently, because partitioning of the problem space is non-trivial; it might even change dynamically as the simulation progresses. As mentioned as part of the architecture overview in Sect. 1, we believe that support for load balancing should be provided at the system level, thereby freeing application developers from the burden of managing a dynamic system.

Taskifying MPI Applications may need to be (re-)balanced due to inherent load imbalances or to support shrinking and expanding the set of nodes assigned to

the application. In the Phase 1 report [59] of this project, we proposed to *taskify* MPI processes using oversubscription, which results in multiple *Tasks* (i.e., MPI processes) per core that can be migrated transparently without application code modifications. MosiX-like algorithms [2] can be used for this kind of system-level load balancing in case communication between application tasks is insignificant. In other cases, repartitioning methods that consider the task communication graph are required [57].

Requirements for Efficient Load Balancing In the context of our system architecture, the requirements for a load-balancing method are: (1) effective load balancing with low inter-node communication (edge-cut), (2) low amount of migration to reduce MPI process migration costs, and (3) low overhead of the method itself. The method's input should be the local node's view of the weighted task communication graph, which can be obtained by monitoring the application at the OS/MPI level. Since graph partitioners, like ParMetis [52], are known to be computationally expensive, we developed a nearest-neighbor diffusion-based repartitioning method.

Method Description The method consists of four main phases that require point-to-point communication between neighbor nodes only and, with the exception of the flow calculation, have *O(number of tasks per node)* computational complexity.

1. Each node **determines** its **neighbor nodes** from the task communication graph.
2. **Flow calculation:** Computation of minimal load flows between neighbor nodes that lead to global balance using 2nd-order diffusion [13, 45]. The diffusion is stopped between each node pair individually if the load flow of an iteration falls below a specified threshold. The required number of iterations grows with the number of nodes. However, with low-latency networks the observed run time is within the low millisecond range even with thousands of nodes [39].
3. **Task selection:** Tasks at the partition border are selected for migration to realize workload flows using different weighted criteria to achieve low edge-cut [14].
4. **Partition refinement:** Edge-cut is improved with a parallel KL/FM-based refinement algorithm [58] that smoothes partition borders by swapping weighted tasks between neighbor node pairs independently within a certain imbalance tolerance. If the task selection result was not within the tolerance, the optimization goal is "balance" instead of "edge-cut".

Evaluation Workloads We implemented the diffusion method within the Zoltan load balancing library [12] and evaluate the performance on Taurus in a normal, non-oversubscribed setup. MPI processes own multiple migratable user-level tasks. Two scenarios with 3D task meshes are used for performance evaluation: the *Cloud scenario* consists of computation time measurements from $36 \times 36 \times 48$ grid cells over 100 time steps of COSMO-SPECS+FD4 [38] and the synthetic *Shock scenario* simulates the evolution of a spherical shock wave over a $160 \times 80 \times 80$ grid with a four times increased workload at the wave front over 169 time steps. Figure 8 shows the workload distribution for a 2D version of the shock scenario with two examples of resulting partitionings.

Fig. 8 Left: workload distribution for a selected time step of the 2D shock scenario, red indicates 4 times increased workload; Center: Resulting partitioning for 96 processes with ParMetis, colors represent individual partitions; Right: Resulting partitioning with Diffusion

Diffusion Results We compare our method with four other (re)partitioning methods: a hierarchical space-filling-curve method (FD4/SFC) [37], recursive coordinate bisection and space-filling curve from Zoltan (Zoltan/RCB and SFC), as well as AdaptiveRepart from ParMetis [52] with 0.5% imbalance tolerance. Note, that SFC and RCB are coordinate-based and not graph-based; they require application knowledge about spatial coordinates of tasks. Since diffusive load balancing requires a partitioning to exist, we use Zoltan/RCB to initialize the partitioning. Figure 9 shows the results for different metrics (lower is better), each averaged over the time steps (except median for run time):

- *Load Imbalance (max. load among processes/average load—1):* Diffusion performs better than ParMetis, but worse than the three geometric methods.
- *Migration (max. no. of tasks a process imports and exports/avg. no. of tasks per process):* Diffusion outperforms the other methods, especially with the Shock scenario (factor 3–10 less migration).

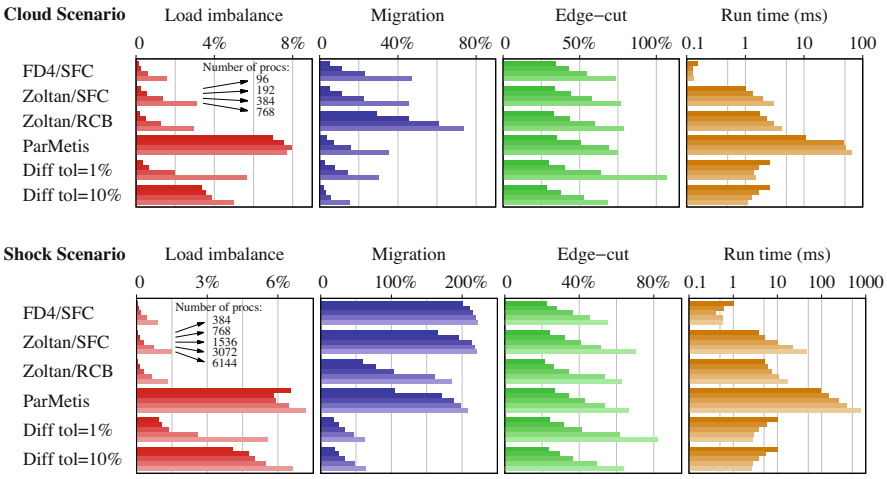

Fig. 9 Evaluation of diffusion-based load balancing with Cloud (62208 tasks) and Shock (ca. 1M tasks) scenarios. Bar shades show results for 96–768 and 384–6144 MPI processes, respectively

- *Edge-cut (max. edge-cut among all processes/avg. number of edges per process):* All methods achieve very similar results, except diffusion with low imbalance tolerance at high process counts (coarse granularity).
- *Run time of the method (max. among all processes):* Due to its scalability, diffusion is clearly faster than the Zoltan methods at higher parallelism and 1–2 orders of magnitude faster than ParMetis. At 6144 processes, it requires 2.5 ms only.

As can be seen, the imbalance tolerance of the diffusion method allows to trade-off load balance vs. edge-cut and migration. We can conclude that our load balancing method enables fast, scalable, and low-migration graph-based repartitioning.

2.6 *Beyond L4: Improving Scalability with* M³ *and* SEMPEROS

We conclude the performance and scalability section of this paper with an outlook on a new kernel architecture suitable for heterogeneous many-core systems: SEMPEROS [25].

M³ SEMPEROS is based on M³ [1], a hardware/software co-designed system architecture. Like L4Re, which we described in Sect. 2.2, these systems are microkernel OSes that manage access rights to all system resources based on *capabilities*. A process can only use a resource, if it owns a capability to it. Such resources include threads, memory allocations, and files, but they are also used to grant and revoke access to CPU cores and the ability to send messages between the threads running on these cores. A key aspect of the M³ design is that the OS kernel runs on just one core, which remotely manages all other cores[2] in the system.

Heterogeneous Architectures Since it is not necessary that all cores of a system can run an OS kernel, M³ is suitable for heterogeneous system architectures with different kinds of CPU cores. The current No. 1 system in the TOP500 list of supercomputers, China's Sunway TaihuLight system [19], is based on such an architecture. Each node has "big" cores capable of running an OS kernel, as well as many small compute cores that are optimized for computation, but lack the architectural support for an OS kernel.

SEMPEROS To put hundreds of cores under the control of a microkernel OS, the kernel and its capability system need to scale. SEMPEROS extends M³ to manage the

[2] M³ provides a hardware abstraction to integrate accelerators in the same way as general-purpose cores. Therefore, we usually use the term *processing element* in an M³ context. HPC workloads can benefit from generalized accelerator support, but it does not influence how the capability system works. In this paper, we therefore use the common term *cores* to simplify the explanation.

system using multiple kernels. The compute cores of the system are organized into as many partitions as there are kernel instances, thereby increasing the total number of cores (and compute threads) the system can handle. Each of the SEMPEROS kernel instances executes on its own privileged core, but they coordinate with each other via a *distributed capability protocol* [25]. The collaboration between cores managed by different kernels is transparent to the applications.

Capability Model SEMPEROS implements partitioned capabilities. The capabilities are stored within a protected address space and the kernel supervises capability operations. Application processes can delegate and revoke capabilities via the respective system calls. The kernel records the delegations in a capability tree that stores the relations between capabilities. Using this capability tree, SEMPEROS implements recursive revocation, by which all access rights that originated from the specific capability can be revoked by deleting this capability and all its descendants.

Distributed Capability Protocol In a distributed capability system, the data structure storing the capability tree is split between multiple kernels. Because the same capabilities can be manipulated simultaneously by different kernels in the system, SEMPEROS kernel instances need to coordinate certain capability operations. We analyzed all possible interleavings of capability operations (e.g., during delegation or revocation) and developed a protocol that prevents inconsistencies in the capability tree. This protocol integrates a confirmation for capability delegation (similar to a two-way handshake) and a distributed mark-and-sweep algorithm to revoke capabilities [25].

Evaluation Setup We evaluated SEMPEROS using the cycle-accurate gem5 simulator [6]. The experiments were run with up to 640 out-of-order cores integrated into a single network-on-chip. The applications used in our evaluation stress the capability system, as they extensively use OS services. In particular, they make heavy use of the M^3/SEMPEROS in-memory file system, which grants access to memory ranges within files by delegating memory capabilities to applications and revoking those capabilities when files are closed again. We assume that similar usage patterns would also occur during checkpointing operations on a future HPC system with storage-class memory in each node.

Scalability Results To quantify the scalability of SEMPEROS, we computed the parallel efficiency, exposing the runtime overhead a single benchmark instance (i.e. process) experiences when it is executed in parallel with a number of identical benchmark instances. Figure 10a depicts an overview of the parallel efficiency of all application benchmarks we evaluated. The scalability of the applications improve when increasing the number of kernels managing the system as depicted in Fig. 10b.

The capability protocol in general is applicable to any distributed capability system implementing global IDs and stores all capability relations in a capability tree.

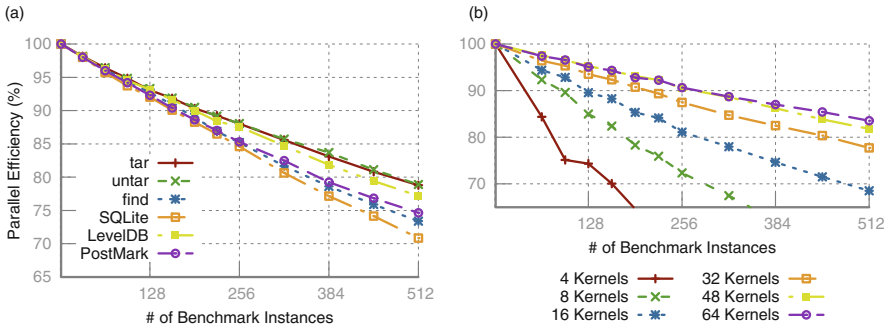

Fig. 10 Scalability evaluation of SEMPEROS. (**a**) Parallel efficiency of application benchmarks. (**b**) Level DB key-value store

3 Algorithms and System Support for Fault Tolerance

In this section we discuss FFMK research results on fault tolerance. Our activities concentrated on two areas: First, we introduce new methods for resilience of MPI applications at scale. To this end, we developed probabilistic and deterministic algorithms for resilience of collective operations (Sects. 3.1 through 3.3). Second, we present new approaches to coordinate and optimize concurrent checkpointing of multiple jobs (Sect. 3.4) and to improve multi-level checkpointing for a single parallel job (Sect. 3.5).

3.1 Resilience in MPI Collective Operations

MPI libraries are usually not fault tolerant and therefore not able to complete a collective operation correctly in case of a fault. Instead, the whole application will either hang or crash. Unfortunately, the overall system reliability decreases as the number of processes involved in the computation grows.

Existing Recovery Approaches The MPI standard does not address resilience of parallel applications, with the exception of return codes for detected errors. Currently, there are multiple outstanding proposals how to address faults during MPI run-time. Most work on this subject is characterized as either *backward recovery* or *forward recovery*. The former tries to restore a correct *past* state, whereas the latter attempts to establish a *new* correct state. For example, ULFM [8] applies forward recovery, entailing a set of tools that allow applications to deal with detected faults.

Fault-Tolerant Algorithms Fault-Tolerant Collective Operations (FTCO) [43] is a new forward recovery approach. FTCO relies on parallel algorithms that apply to tree-based collective operations in MPI. It includes resilient versions of collective operations such as *Bcast*, *Reduce* and *Allreduce*, for any tree topology. This

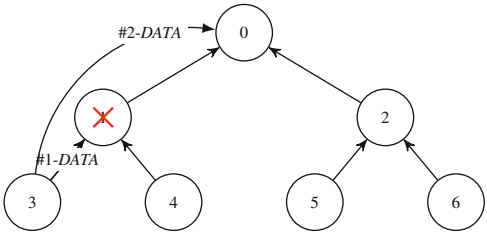

Fig. 11 Message flow after *DATA* timeout: Node #3 overcomes fault detected on node #1 by bypassing it and sending the data to the next node #0 along the tree

algorithm detects faults by a per-node calculated timeout and overcomes faults by excluding failed processes. It is intended for use-cases that can tolerate process faults, such as Monte–Carlo method or PDE solvers. FTCO delivers a result to every live process, so that the application may keep running without handling those faults. The FTCO algorithm minimizes the fault-free performance penalty and shows a small increase in latency and messages per fault, regardless of job size. This offers a transparent and scalable forward recovery alternative to the costly legacy backward recovery mechanisms. For example, Fig. 11 demonstrates a simple case, where node #3 overcomes the fault detected on node #1 by bypassing it and sending the data to the next node (#0) along the tree.

FTCO Results FTCO differs from other approaches, such as ULFM, by localizing the detection and recovery of faults, while other approaches involve the entire group of processes in the MPI job. Our experimental results with the FTCO approach support this claim, showing that the latency with FTCO is proportional to the number of serial faults, regardless of the number of MPI proceeses: Table 2 shows the FTCO latency of the *Allreduce* operation for different combinations of offline faults and increasing number of processes. We chose a timeout of 2 s, which is the default for ULFM [43]. In the table, each figure represents the average longest time (in seconds) for a process to complete the same *Allreduce* call. In a tree topology, multiple faults could be either serial or parallel: parallel faults occur in different subtrees, thus their recovery time may overlap, while serial faults are handled one after the other. We found that two parallel faults take approximately the same amount of time as a single fault; adding a third parallel fault to two serial faults does not change the latency.

Table 2 FTCO: average longest time in seconds for a process to complete *Allreduce* call

No. of process	No faults	1 fault	2 parallel faults	2 serial faults	3 mixed faults
64	0.0005	2.0095	2.0094	4.0194	4.0194
128	0.0006	2.0123	2.0123	4.0221	4.0222
256	0.0007	2.0121	2.0123	4.0220	4.0219
384	0.0017	2.0114	2.0323	4.0404	4.0217
512	0.0024	2.0343	2.0110	4.0421	4.0422

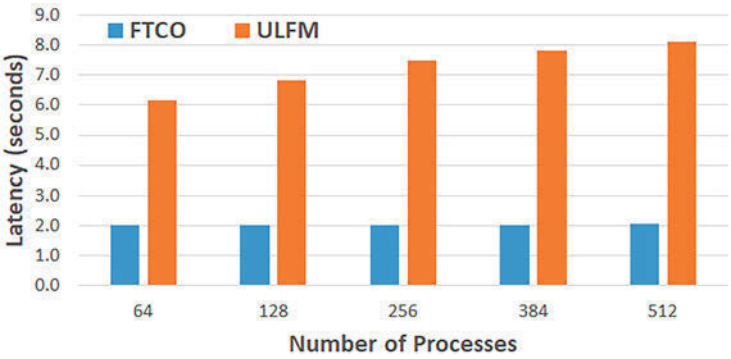

Fig. 12 The latency cost of one fault (in seconds) with FTCO and ULFM: FTCO outperforms ULFM; both approaches use the same fault-detection timeouts and tree topology

FTCO vs ULFM Figure 12 shows a comparison between the latency of FTCO and ULFM for the *Allreduce* operation with a single fault and an increasing number of processes. In both cases we used the same parameters, like fault detection timeouts and tree topology. Each test started by injecting a fault into one process and then attempted an *Allreduce* on multiple communicators with that process.

Integration of FTCO into UCX We developed a prototype library to further measure the performance of FTCO based on UCX. UCX [54] is an open-source point-to-point communication library, optimized for performance on low-latency interconnects. UCX is designed to provide a high-level of abstraction for communication, and to consider the specifics of the local NIC, to find the optimal send and receive methods. UCX queries the capabilities of each NIC to establish which hardware accelerations are present, and chooses the optimal parameters for sending messages, including the NIC, port and protocol (e.g. rendezvous). In order to apply fault-tolerance algorithms to MPI applications, we extended UCX with an implementation of MPI's collective operations. Our library is suitable for applications that can withstand partial failures, where the overall result is not effected by some faults, or where the application can overcome them.

Our extension contains a basic, deterministic implementation for fault-free collective operations. It demonstrated the benefit of a persistent collective operation: applications often repeat the same collective operation call, and so the library can reuse past structures instead of creating it from scratch for each call. Our UCX-based library can be used with any MPI implementation, and provides a foundation for further experimental results and optimizations.

3.2 Corrected Gossip Algorithms

An alternative to the deterministic approach taken with FTCO is to use randomized gossip-based algorithms, which have been shown to yield better recovery latency for some applications. Gossip algorithms have been widely successful in various contexts that did not require strong consistency. Yet, they become rapidly inefficient once about 50% of the nodes were reached, because messages are more likely to be sent to nodes that have already received a message before.

Gossip-Based Broadcast Let us consider how broadcast among processes can be implemented using gossip. The root sends the broadcast payload to a random subset of other processes. When a new process receives the payload and gets *colored*, it starts sending messages to randomly selected processes as well. This dissemination runs for a fixed period of time, after which all colored processes have received the broadcast payload.

Closing the Gaps The gossip phase attempts to color as many processes as possible, but due to potential failures and the random nature of gossip, the protocol cannot guarantee that all live processes are actually colored. Our new algorithm therefore enters a deterministic correction phase, in which it tries to color these remaining processes. For correction, all processes are reorganized into a virtual ring (e.g., according to their MPI rank numbers from 0 to $P-1$ for P processes). On this ring, uncolored processes create *gaps*, where the *maximum gap size* is the length of the longest sequence of uncolored processes. All colored processes now send messages to their neighbors on the ring, thereby closing the gaps with few messages per node.

Corrected Gossip Algorithms We designed three different protocols based on this idea of combining randomized and deterministic algorithms for improving the broadcast latency [27]. These three algorithms allow us to choose various tradeoffs between consistency, simplicity, and performance. The three algorithms are: (1) *opportunistic*, which applies the correction without checking for completion; (2) *checked*, which runs the correction until all nodes received the message, provided that no nodes fail during the correction; and (3) *fail-proof*, which applies the correction and guarantees that all active nodes receive the message, provided that no more than f nodes fail during the operation. Our algorithms do not require multicast, failure detectors, timeouts, acknowledgments, or reconfiguration procedures. The result of this work is the "corrected-gossip" paradigm [27].

Framework for Collective Algorithms Based on the corrected-gossip paradigm, we also developed a framework for failure-proof collective operations that generates online an independent spanning tree. Generation can be completed successfully even with an arbitrary number of active nodes and up to f online failures. Based on the system's mean time between failures (MTBF), an appropriate value f could be chosen as the maximal number of faults supported by the algorithm. Compared to alternative methods for fault recovery, this approach allows a trivial

recovery procedure, provided that sufficient spanning trees are maintained during the application run.

3.3 Corrected Tree Algorithms

The reliability properties of all correction schemes of "corrected gossip" have been proven [27]. However, due to being probabilistic, the gossip algorithm used in the dissemination phase needs to send more messages than an optimal tree-based broadcast algorithm in order to color nodes. Tree-based dissemination will, however, miss a large number of processes, if any process close to the tree's root fails. In general, failure of any non-leaf process in the tree results in all its descendants remaining uncolored.

Can We Save the Trees? We developed a *corrected tree* algorithm which splits communication among processes into two phases, exactly like we did with corrected gossip: *dissemination* and *correction*. The idea is to correct the result of a failed tree-based broadcast during the dissemination phase. With failure-proof correction, full coloring can be guaranteed even if processes fail during the broadcast [27].

Requirements for Corrected Trees The goal of the broadcast is to guarantee information propagation from the root process to every live process, even if some processes fail. In a broadcast operation among P processes, the *root* process propagates a message reliably to all other processes. Without loss of generality we assume the root process to have rank 0, other processes have ranks $1, \ldots, P-1$. To ensure that correction can color uncolored processes quickly, the maximum gap size ought to be small. A tree maintaining such a property should have its subtrees spread across the correction ring to avoid the danger of having uncolored processes cluster together on the ring. For lowering correction latency, multiple small gaps are better than few large ones.

Interleaved Trees The key idea behind corrected tree algorithms is that nodes of the tree can be renumbered in such a way that parent and child nodes do not become neighbors on the ring. Instead, nodes from a subtree below a failed node shall always have a close neighbor from outside their own subtree, so the they can be colored in the correction phase. In this paper, we only describe how to interleave k-ary trees. However, the scheme is also available for Lamé trees, which include Binomial trees. A more detailed description is given in the original publication [31].

Interleaving k-ary Trees Given the root process at level 0, a full k-ary tree has k^ℓ processes at level ℓ, and k^ℓ subtrees that have their root process at that level. The processes in these subtrees can have a distance of k^ℓ in the ring. Process r has the child processes r':

$$\{r' \mid r' = r + i \cdot k^\ell, 0 < i \leq k, r' < P\}$$

This interleaving ensures that a failing process on level ℓ leads to every k^ℓ-th process being uncolored. Thus, $f = k^\ell - 1$ failures on level ℓ or below can be tolerated, and still every k^ℓ-th process will be colored after the dissemination.

Correction Phase The correction phase follows the dissemination phase and is independent of the tree type. It ensures coloring of the processes that the dissemination phase left uncolored due to failures. All three correction algorithms developed for corrected gossip [27] are directly applicable.

Simulation Results In our simulation-based evaluation, we used the same fail-stop fault model as in the corrected-gossip publication [27]. During broadcast, every process is either dead or alive. A process either sends all messages required by the protocol or none at all, but failures can occur anywhere outside of the broadcast operation. The simulation is based on the LogP-model [10]. The graph in Fig. 13 shows the number of messages sent for different numbers of faults. All tree-based broadcasts send fewer messages than corrected gossip and are thus more efficient with regard to this metric.

Latency Measurements To measure the average latency of a broadcast, we implemented all algorithms using MPI and ran them on the Piz Daint supercomputer installed at ETH Zurich. The results shown in Fig. 14 therefore include overhead due to the physical properties of the interconnect. Note that there are no faults in this experiment, so we can compare our implementations with the broadcast implementation from Cray. Since our MPI-based implementation cannot use the shared-memory optimization the Cray algorithm uses, we include performance

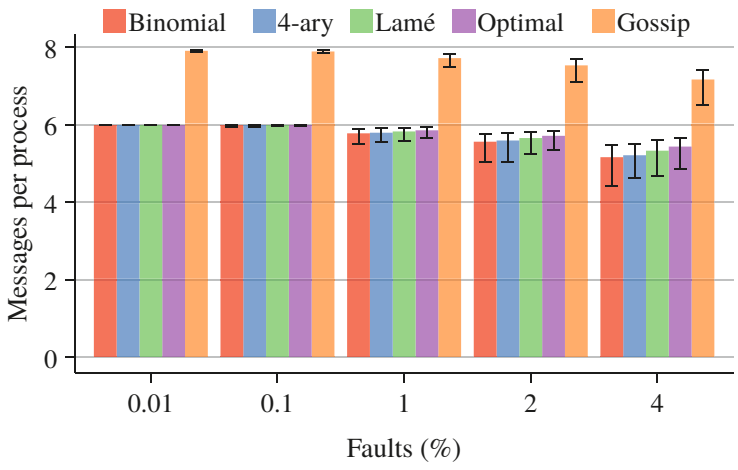

Fig. 13 Average number of messages sent per process in presence of failures; four different corrected tree algorithms and corrected gossip were simulated with varying percentage of failed processes

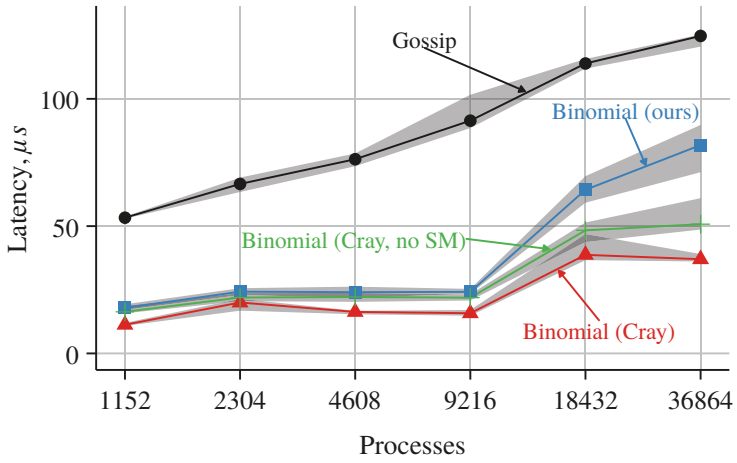

Fig. 14 Broadcast median latency of corrected tree, corrected gossip, and vendor-provided implementations of broadcast without any failures

of the Cray algorithm with shared memory disabled for reference (green line in diagram).

We find *Corrected Trees* to be a simple, yet efficient protocol for fault-tolerant collective group communication. A more detailed evaluation can be found in the publication [31].

3.4 Checkpointing Scheduling

Only some applications can benefit from forward recovery by tolerating process or node failures. All other HPC codes rely on system-level or application-assisted fault-tolerance services. Periodic checkpoint/restart (C/R) [11, 65] is an effective mechanism to alleviate failures during execution of HPC applications, which often require a vast number of nodes for a relatively long time [15]. The overhead of checkpointing should be kept as small as possible, as typically the computation has to be paused during a checkpoint write to store a consistent state of the application. Furthermore, on a large supercomputer, which is shared by several parallel jobs, checkpointing should be coordinated between applications to avoid performance bottlenecks. Otherwise, as the storage is typically shared between all applications, concurrent checkpoint writes would unnecessarily delay the computation tasks.

Uncoordinated Checkpointing Hurts As part of the FFMK project, we designed a mechanism to coordinate concurrent checkpoints of large applications running on a supercomputer and sharing a dedicated burst-buffer [42] or parallel file system (PFS) [41] for checkpointing [22]. We assume that the system executes multiple

parallel applications at the same time and that it provides a single shared PFS (e.g., Lustre, GPFS, ...) for all applications. Typically, each application uses fixed checkpoint intervals calculated independently of the other applications based on Daly's method [11]. However, Daly assumes a constant checkpoint write duration during the application's lifetime, whereas in case of concurrent checkpointing of different applications checkpoint costs vary and depend on other applications' activities due to limited write bandwidth and other interferences.

Checkpoint Scheduling Approach In our work, we take this variation into account and rearrange the checkpoints of jobs, by predicting conflicts, in order to minimize the overall system resource usage. This is achieved by *scheduling checkpoints* earlier or later than originally planned. The decision whether to postpone a checkpoint to another timeslot is based on a constructed optimization problem for the predicted conflicts. By minimizing the constructed problem, non-overlapping timeslots to write checkpoints are found and applied.

Checkpoint Scheduling Architecture While system-level checkpointing takes advantage of system-wide information not visible to jobs such as mean-time to failure (MTTF), bandwidth, or other jobs' activities, it suffers from unnecessarily large checkpoints (higher checkpoint costs). On the other hand, user-defined checkpointing is done within the application which uses a runtime library for checkpointing. The library remembers the important data to be saved and also knows at which time checkpoints should be written (application-level information). This way the approach reduces the checkpoint size while lacking the system-wide information. In our work, we combined system-level with user-defined checkpointing in order to leverage the advantages of both methods [22]. The coordination among different jobs is performed by a coordinator service running on the system side. The service has access to system-wide information like MTTF, the bandwidth of shared storage systems, and the activities of other jobs (system-level checkpointing).

User-Level Library Support To achieve this, we modified the SCR library [44] to provide it with the ability to interact with our coordinator service, which instructs SCR when to perform checkpoint requests. The application uses the library to write checkpoints (user-level checkpointing). This library cooperates with the coordinator-service, whereby whenever a checkpoint call is made by the application (which is done frequently and periodically), a request is sent to the coordinator and if accepted, the write will be performed by the library. Otherwise, the checkpoint call will be ignored and the application continues with the computation.

System-Level Coordinator The decisions on the coordinator side are based on solved optimization problems constructed for the predicted conflicts. Then the incoming checkpoint requests are either accepted or rejected based on the computed decisions to get optimal checkpoint times.

Figure 15 shows that we are able to reduce the C/R overhead by up to 20% by coordinating checkpoints compared to state-of-the-art approaches.

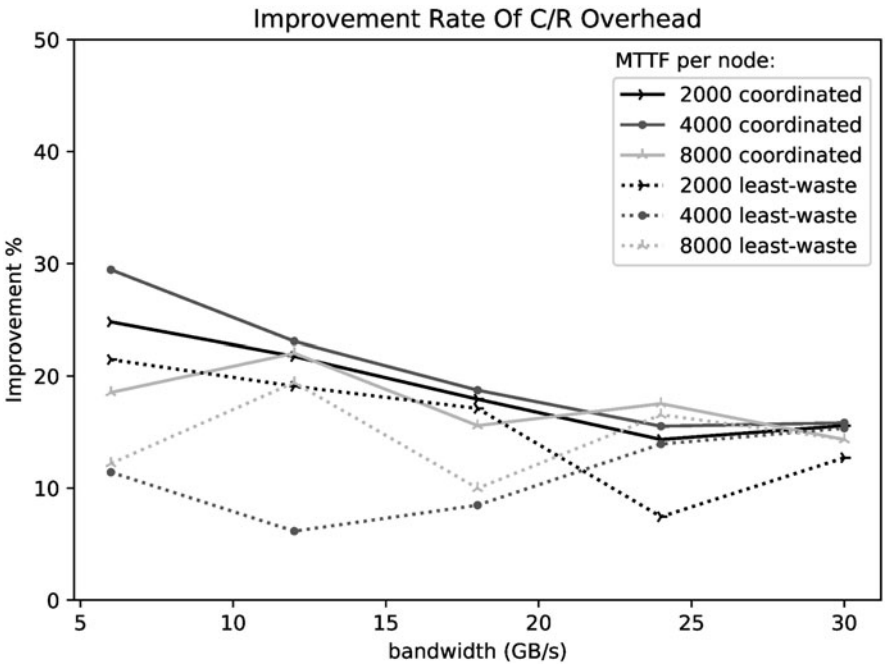

Fig. 15 Improvement rate of C/R overhead

3.5 Multi-Level Checkpoint/Restart

To further strengthen the C/R mechanism of the FFMK OS platform, *multi-level checkpoint/restart* is used as a solution to address a wide range of failures of different severity occurring in large supercomputers [21]. Severe failures require application state to be recovered from the most stable storage devices. However, this comes at high cost, as stable storage devices are slower to write to than transient storage. This is the key motivation of using multi-level checkpoint/restart.

Component Failure Rates To fully grasp the multi-level hierarchy of failures and storage devices, we constructed a comprehensive model of a typical large cluster containing nodes, local storages (ramdisk, SSD), network switches, power supplies, different shared storage systems (burst-buffer, PFS), etc. Additionally, we modeled 5 different levels of checkpoints, all of which are supported by the SCR [44] library and most of them are also available on other checkpointing libraries as well (FTI [4], VeloC [46]):

- **Local Level:** The first checkpoint level is the node's local storage (ramdisk, SSD). While this level benefits from the lowest checkpointing cost among all levels, it cannot survive fatal faults where the node becomes unavailable.

- **XOR Level:** The next level is the XOR level, at which for each node a parity segment is computed and distributed among a set of nodes (XOR group) according to [24, 47]. Each node stores its local checkpoint along with the XOR parity data. This level survives a single node of an XOR group becoming unavailable.
- **Partner Level:** A more stable level is the partner scheme, where the checkpoint is stored in the node's local storage along with the partner node's storage (two copies). This level fails to restart the job, if a node and its partner fail together.
- **Shared Levels:** The most stable levels use shared storage (among all compute nodes), namely the burst buffer and the parallel file system. However, they provide lower checkpointing bandwidth per node, because many (or all) nodes may access them concurrently.

Modeling Failures Per Component To have a proper estimation of the stability of different checkpoint levels (i.e., how often they fail), we studied a wide range of possible failures occurring on supercomputers and investigated their effect on each checkpoint level. Using the estimations, we defined the optimum checkpoint intervals for each level minimizing the application's lifetime. To study different types of failures, we differentiated light faults and fatal failures, where after a light fault the node stays alive and the job can be restarted from the local storage device. A fatal failure makes a node or a set of nodes unavailable (node down). In this case, the job must be restarted from more stable levels (XOR, partner, etc.). Additionally, we took correlations into account by arranging the nodes of a cluster in different correlated groups (network, power supply, etc.), whereby the nodes belonging to the same correlated group are vulnerable to failures of the same origin making all nodes in the group unavailable at the same time (e.g., a network switch outage). Figure 16 visualizes, how this correlation of nodes translates into a correlation graph.

In addition to the compute nodes, we considered I/O nodes (burst-buffer) which are also vulnerable to failures making the burst-buffer unavailable for a restart. Although such faults do not interrupt the job's computation, losing checkpoints on the burst-buffer may impact the job's lifetime. Another case of a checkpoint's

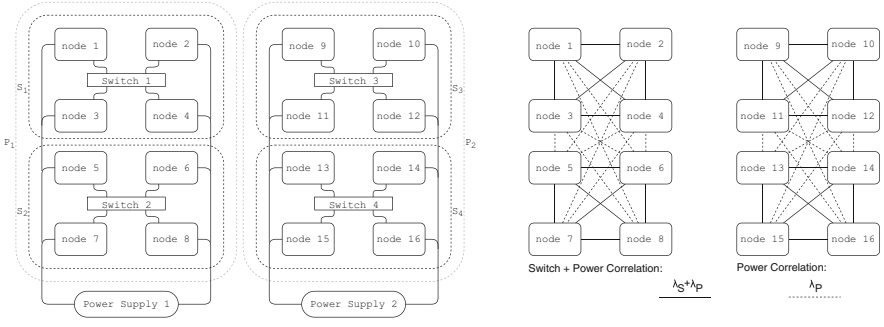

Fig. 16 Correlation of nodes in a cluster

unavailability occurs, if due to a node down, the job is restarted from the XOR level and the new node allocated to the job lacks the partner level checkpoint. Hence, in case of another failure, an attempt to restore from partner-level will fail to recover the just restarted job a second time until the next partner checkpoint is written.

Model-Based Adaptation of Checkpoint Behavior Finally, we introduced a mechanism to model the expected lifetime of applications writing multi-level checkpoints on a given set of checkpoint levels considering all investigated types of failures. Using this model, we properly choose the optimal set of checkpoint levels for each job minimizing the expected lifetime.

Multi-Level Checkpoint Architecture Our implementation of multi-level checkpoint/restart consists of a system-level daemon and a library that is linked into the application. The daemon communicates with the library in the application using TCP sockets and directs the local checkpoints (written by the application) to different checkpoint levels transparent to the application. It computes the optimum checkpoint intervals of each level using system-wide information and application-specific parameters. For each checkpoint call made through the library, the daemon determines the optimal level of the checkpoint (or simply rejects it). We employ the following optimizations and automations in our design to further improve performance and reliability:

- **Asynchronous Checkpointing:** To further reduce the checkpointing costs of the lower levels, checkpoint writes are performed asynchronously. The application writes its state to the node's local storage, providing fast write performance. Thereafter, the operation of checkpointing at the lower levels (e.g., partner, burst buffer, PFS) is performed by the daemon in the background while the application returns to the computation. During the background operation of the daemon, every incoming request for checkpointing to the same level from the job is ignored until the persisting is finished.
- **Fast Recovery:** To ensure the fast recovery in case of light failures (node alive), ULFM [7] is used to provide the capability of automatically recovering the failed ranks with the most recent checkpoint and reordering the ranks to the original arrangement, transparent to the developer, application, and the resource manager.
- **Job Life Cycle and Recovery** When a new job is started, a short-lived controller service is executed by the first rank on each node (i.e., the node's local rank 0). This service forks a daemon process for each rank on the node, connects them to the corresponding ranks, and then terminates. The per-rank daemons are responsible for sending status information and receiving instructions. The per-rank daemons stay alive until the application is finished with the computation. If a rank fails, its daemon terminates and the rank is later restarted using ULFM. When a rank is restarted, or when it notices that its daemon died, it will execute the previously mentioned controller service again, which will then reconnect a new daemon process with the rank. This is done transparently to the application and the existence of both daemons and controller services remains transparent to the user and the developer.

- **Rotating Partner Nodes:** To further enhance the stability of the partner level, a novel approach will be engaged in which there are no fixed partners for the nodes, instead, the partner node of a specific node is changed on each turn and chosen by the daemon. This allows reducing the probability of unavailability of checkpoints at this level. In addition, partner checkpoints and data transfers will be performed by service daemons of both nodes using Remote Direct Memory Access (RDMA) in order to have fast partner checkpoints in the background.

Our evaluations of the introduced multi-level checkpoint/restart model (the failure rates and optimal intervals) show that we managed to reduce the C/R overhead of jobs by up to 10% in the investigated cases (50,000 nodes, 3–5 levels, 6–240 GB checkpoints) compared to the state-of-the-art approach. Our observations indicate that the overhead is reduced further as the number of levels or the checkpoint size increases.

4 Conclusions

The FFMK project had the very ambitious goal of creating a new operating-system (OS) platform for Exascale computing. Unfortunately, we did not reach the milestone of a fully integrated prototype at the end of phase 2. However, several essential building blocks have been adapted for HPC and new algorithms and services have been developed to meet the challenges we expect from these upcoming extreme-scale systems.

Multi-Kernel Node OS At the node level, the mature L4Re microkernel together with L^4Linux enable the decoupled-thread model, which combines noise-free execution and full Linux compatibility for applications. We studied and quantified the influence of hardware performance variation, a source of noise that still remained. Furthermore, we gave an outlook on how to manage OS-level resources at extreme scales of parallelism with a kernel architecture for scalable capability management.

Dynamic Platform Management Although mostly an activity from the first three-year phase of the project, we continued research on dynamic platform management at the system level. We researched fault-tolerant gossip-based algorithms for obtaining load and health information from nodes. We integrated these information dissemination algorithms with a per-node management daemon and a distributed decision making module. However, one component of the FFMK architecture's load balancing service remained elusive: migration of processes has been shown to work at small scale for simple InfiniBand-based programs, but robust C/R-based migration is not available yet. In Phase 2, we developed highly efficient diffusion-based load balancing algorithms; they complement the architecture and can already be used for application-level load balancing, as they have been integrated into a widely-used framework.

Fault Tolerance The main activities of the second phase of this project concentrated on efficient and fault-tolerant communication, as well as system-level support for reducing checkpoint overhead. Fault-tolerant collective operations (FTCO) have been developed as a drop-in replacement for MPI libraries and were prototyped in the UCX communication library. As a potentially more efficient alternative, we introduced a new class of two-step algorithms based on either gossip or trees: Corrected Gossip and Corrected Trees enable inherently fault-tolerant collective operations for applications that can continue even after some processes failed. Globally optimized checkpoint scheduling and multi-level checkpointing minimize checkpoint/restart overhead for all other applications. The latter optimizes the cost of checkpointing based on a model that includes expected failure rates of components, a failure-correlation graph, and available write bandwidth across the storage hierarchy.

Together, all newly developed algorithms, system services, and low-level OS components form the basis of a fast, scalable, and fault-tolerant HPC operating system for Exascale computing.

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 [17]. 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, the John von Neumann Institute for Computing, and the Swiss National Supercomputing Centre (CSCS) for providing compute time on their supercomputer systems.

References

1. Asmussen, N., Völp, M., Nöthen, B., Härtig, H., Fettweis, G.: M3: A hardware/operating-system co-design to tame heterogeneous manycores. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2016)
2. Barak, A., Guday, S., Wheeler, R.: The MOSIX Distributed Operating System: Load Balancing for UNIX. Lecture Notes in Computer Science, vol. 672. Springer, Berlin (1993)
3. Barak, A., Drezner, Z., Levy, E., Lieber, M., Shiloh, A.: Resilient gossip algorithms for collecting online management information in exascale clusters. Concurr. Comput. Pract. Exp. **27**(17), 4797–4818 (2015)
4. Bautista-Gomez, L.A., et al.: FTI: high performance fault tolerance interface for hybrid systems. In: SC'11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 32:1–32:32 (2011). http://doi.acm.org/10.1145/2063384.2063427
5. Beckman, P., Iskra, K., Yoshii, K., Coghlan, S.: The influence of operating systems on the performance of collective operations at extreme scale. In: 2006 IEEE International Conference on Cluster Computing, pp. 1–12 (2006). https://doi.org/10.1109/CLUSTR.2006.311846
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The Gem5 simulator. SIGARCH Computer Architecture News (2011)
7. Bland, W.: User level failure mitigation in MPI. In: Euro-Par 2012: Parallel Processing Workshops - BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience,

UCHPC, VHPC, Rhodes Islands, August 27–31, 2012. Revised Selected Papers, pp. 499–504. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-36949-0_57
8. Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.J.: An evaluation of user-level failure mitigation support in MPI. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) Recent Advances in the Message Passing Interface, pp. 193–203. Springer, Berlin (2012)
9. Cavium: ThunderX_CP Family of Workload Optimized Compute Processors (2014). https://www.marvell.com/content/dam/marvell/en/public-collateral/server-processors/marvell-server-processors-thunderx-cp-product-brief.pdf
10. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., Von Eicken, T.: LogP: towards a realistic model of parallel computation. In: Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 1–12. ACM, New York (1993). https://doi.org/10.1145/155332.155333
11. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. Future Gener. Comput. Syst. **22**(3), 303–312 (2006). https://doi.org/10.1016/j.future.2004.11.016
12. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan data management services for parallel dynamic applications. Comput. Sci. Eng. **4**(2), 90–97 (2002)
13. Diekmann, R., Frommer, A., Monien, B.: Efficient schemes for nearest neighbor load balancing. Parallel Comput. **25**(7), 789–812 (1999)
14. Diekmann, R., Preis, R., Schlimbach, F., Walshaw, C.: Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. Parallel Comput. **26**(12), 1555–1581 (2000)
15. Feinberg, A.: An 83,000-processor supercomputer can only match 1% of your brain (2013). http://gizmodo.com/an-83-000-processor-supercomputer-only-matched-one-perc-1045026757
16. Ferreira, K.B., Bridges, P., Brightwell, R.: Characterizing application sensitivity to OS interference using Kernel-level noise injection. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, pp. 19:1–19:12. IEEE Press, Piscataway (2008). http://dl.acm.org/citation.cfm?id=1413370.1413390
17. FFMK Website. http://ffmk.tudos.org. Accessed 5 Aug 2019
18. Forum, M.P.I.: MPI: a message-passing interface standard. Standard 3.1, University of Tennessee, Knoxville (2015)
19. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 072001 (2016). https://doi.org/10.1007/s11432-016-5588-7
20. Gerofi, B., Takagi, M., Hori, A., Nakamura, G., Shirasawa, T., Ishikawa, Y.: On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1041–1050 (2016). https://doi.org/10.1109/IPDPS.2016.80
21. Gholami, M., Schintke, F.: Multilevel checkpoint/restart for large computational jobs on distributed computing resources. In: 38th Symposium on Reliable Distributed Systems (SRDS'19) (2019)
22. Gholami, M., Schintke, F., Schütt, T.: Checkpoint scheduling for shared usage of burst-buffers in supercomputers. In: The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, August 13–16, 2018, pp. 44:1–44:10. ACM, New York (2018). https://doi.org/10.1145/3229710.3229755
23. Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a lightweight supercomputer Kernel: lessons learned from Blue Gene's CNK. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2010). https://doi.org/10.1109/SC.2010.22
24. Gropp, W.D., et al.: Providing efficient I/O redundancy in MPI environments. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 3241, pp. 77–86 (2004). https://doi.org/10.1007/978-3-540-30218-6_17

25. Hille, M., Asmussen, N., Bhatotia, P., Härtig, H.: SemperOS: A distributed capability system. In: 2019 USENIX Annual Technical Conference (ATC) (2019)
26. Hoefler, T., Schneider, T., Lumsdaine, A.: Characterizing the influence of system noise on large-scale applications by simulation. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/SC.2010.12
27. Hoefler, T., Barak, A., Shiloh, A., Drezner, Z.: Corrected gossip algorithms for fast reliable broadcast on unreliable systems. In: International Parallel and Distributed Processing Symposium, IPDPS, pp. 357–366. IEEE Computer Society, Washington (2017). https://doi.org/10.1109/IPDPS.2017.36
28. IBM: Design of the IBM Blue Gene/Q Compute chip. IBM J. Res. Develop. **57**(1/2), 1:1–1:13 (2013). https://doi.org/10.1147/JRD.2012.2222991
29. Intel: Intel xeon processor E5-1600/E5-2600/E5-4600 v2 product families (2014). https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-2-datasheet.html
30. Kelly, S.M., Brightwell, R.: Software architecture of the light weight kernel, Catamount. In: Cray User Group, pp. 16–19 (2005)
31. Küttler, M., Planeta, M., Bierbaum, J., Weinhold, C., Hä rtig, H., Barak, A., Hoefler, T.: Corrected trees for reliable group communication. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP'19, pp. 287–299. ACM, New York (2019). http://doi.acm.org/10.1145/3293883.3295721
32. Lackorzynski, A., Weinhold, C., Härtig, H.: Combining predictable execution with full-featured commodity systems. In: Proceedings of OSPERT2016, the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, OSPERT 2016, pp. 31–36 (2016)
33. Lackorzynski, A., Weinhold, C., Härtig, H.: Decoupled: Low-effort noise-free execution on commodity system. In: Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS'16. ACM, New York (2016)
34. Lackorzynski, A., Weinhold, C., Härtig, H.: Predictable low-latency interrupt response with general-purpose systems. In: Proceedings of OSPERT2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, OSPERT 2017, pp. 19–24 (2017)
35. Lawrence Livermore National Laboratory: The FTQ/FWQ benchmark. https://asc.llnl.gov/sequoia/benchmarks/FTQ_summary_v1.1.pdf
36. Levy, E., Barak, A., Shiloh, A., Lieber, M., Weinhold, C., Härtig, H.: Overhead of a decentralized gossip algorithm on the performance of HPC applications. In: Proceedings of ROSS'14, pp. 10:1–10:7. ACM, New York (2014)
37. Lieber, M., Nagel, W.E.: Highly scalable sfc-based dynamic load balancing and its application to atmospheric modeling. Future Gener. Comput. Syst. **82**, 575–590 (2018)
38. Lieber, M., Grützun, V., Wolke, R., Müller, M.S., Nagel, W.E.: Highly scalable dynamic load balancing in the atmospheric modeling system COSMO-SPECS+FD4. In: International Workshop on Applied Parallel Computing PARA 2010: Applied Parallel and Scientific Computing 2010. Lecture Notes in Computer Science, vol. 7133, pp. 131–141. Springer, Berlin (2012)
39. Lieber, M., Gößner, K., Nagel, W.E.: The potential of diffusive load balancing at large scale. In: Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016), pp. 154–157 (2016)
40. Liedtke, J.: On micro-kernel construction. In: SOSP'95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pp. 237–250. ACM Press, New York (1995). http://doi.acm.org/10.1145/224056.224075
41. Ligon, W.B., Ross, R.B.: Implementation and performance of a parallel file system for high performance distributed applications. In: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing (HPDC), pp. 471–480 (1996). https://doi.org/10.1109/HPDC.1996.546218

42. Liu, N., et al.: On the role of burst buffers in leadership-class storage systems. In: Proceedings of the 2012 IEEE Conference on Massive Data Storage (MSST), pp. 1–11 (2012). https://doi.org/10.1109/MSST.2012.6232369
43. Margolin, A., Barak, A.: Tree-based fault-tolerant collective operations for MPI. In: Workshop on Exascale MPI (ExaMPI) (2018)
44. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC) pp. 1–11 (2010). https://doi.org/10.1109/SC.2010.18
45. Muthukrishnan, S., Ghosh, B., Schultz, M.H.: First and second order diffusive methods for rapid, coarse, distributed load balancing. Theory Comput. Syst. **31**, 331–354 (1998)
46. Nicolae, B., et al.: Veloc: Very low overhead checkpointing system. https://veloc.readthedocs.io/en/latest/
47. Patterson, D.A., et al.: A case for redundant arrays of inexpensive disks (RAID). In: ACM SIGMOD Record, pp. 109–116 (1988). http://doi.acm.org/10.1145/50202.50214
48. Pedretti, K.T., Levenhagen, M., Ferreira, K., Brightwell, R., Kelly, S., Bridges, P., Hudson, T.: LDRD final report: a lightweight operating system for multi-core capability class supercomputers. Technical report SAND2010-6232, Sandia National Laboratories (2010)
49. Petrini, F., Kerbyson, D., Pakin, S.: The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the 15th Annual IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Anaylsis (SC'03) (2003)
50. Riesen, R., Brightwell, R., Bridges, P.G., Hudson, T., Maccabe, A.B., Widener, P.M., Ferreira, K.: Designing and implementing lightweight kernels for capability computing. Concurrency and Computation: Practice and Experience **21**(6), 793–817 (2009). http://dx.doi.org/10.1002/cpe.v21:6
51. Riesen, R., Maccabe, A.B., Gerofi, B., Lombard, D.N., Lange, J.J., Pedretti, K., Ferreira, K., Lang, M., Keppel, P., Wisniewski, R.W., Brightwell, R., Inglett, T., Park, Y., Ishikawa, Y.: What is a lightweight kernel? In: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS. ACM, New York (2015). https://doi.org/10.1145/2768405.2768414
52. Schloegel, K., Karypis, G., Kumar, V.: A unified algorithm for load-balancing adaptive scientific simulations. In: Proceedings of the IEEE/ACM SC2000 Conference, pp. 59–59 (2000)
53. Seelam, S., Fong, L., Tantawi, A., Lewars, J., Divirgilio, J., Gildea, K.: Extreme scale computing: modeling the impact of system noise in multicore clustered systems. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS) (2010). https://doi.org/10.1109/IPDPS.2010.5470398
54. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., Bouteiller, A.: UCX: an open source framework for HPC network APIs and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43 (2015)
55. Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A., Ishikawa, Y.: Interface for Heterogeneous Kernels: a framework to enable hybrid OS designs targeting high performance computing on manycore architectures. In: 21th International Conference on High Performance Computing, HiPC (2014)
56. Sodani, A.: Knights landing (KNL): 2nd generation intel xeon phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS), pp. 1–24 (2015). https://doi.org/10.1109/HOTCHIPS.2015.7477467
57. Teresco, J.D., Devine, K.D., Flaherty, J.E.: Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In: Numerical Solution of Partial Differential Equations on Parallel Computers. Lecture Notes in Computational Science and Engineering, vol. 51, pp. 55–88. Springer, Berlin (2006)

58. Walshaw, C., Cross, M.: Jostle – multilevel graph partitioning software: an overview. In: Mesh Partitioning Techniques and Domain Decomposition Methods, chap. 2, pp. 27–58 (2007)
59. Weinhold, C., Lackorzynski, A., Bierbaum, J., Küttler, M., Planeta, M., Härtig, H., Shiloh, A., Levy, E., Ben-Nun, T., Barak, A., Steinke, T., Schütt, T., Fajerski, J., Reinefeld, A., Lieber, M., Nagel, W.E.: FFMK: a fast and fault-tolerant microkernel-based system for exascale computing. In: Bungartz, H.J., Neumann, P., Nagel, W.E. (eds.) Software for Exascale Computing - SPPEXA 2013–2015, pp. 405–426. Springer, Cham (2016)
60. Weinhold, C., Lackorzynski, A., Härtig, H.: FFMK: an HPC OS based on the L4Re Microkernel. In: R.W. Wisniewski, B. Gerofi, R. Riesen, Y. Ishikawa (eds.) Operating Systems for Supercomputers and High Performance Computing. Springer Singapore (2019)
61. Weisbach, H., Gerofi, B., Kocoloski, B., Härtig, H., Ishikawa, Y.: Hardware performance variation: a comparative study using lightweight kernels. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) High Performance Computing, pp. 246–265. Springer, Cham (2018)
62. Wende, F., Steinke, T., Reinefeld, A.: The impact of process placement and oversubscription on application performance: a case study for exascale computing. In: Gray, A., Smith, L., Weiland, M. (eds.) Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC 2015, pp. 13–18 (2015)
63. Wisniewski, R.W., Inglett, T., Keppel, P., Murty, R., Riesen, R.: mOS: an architecture for extreme-scale operating systems. In: Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'14), pp. 2:1–2:8. ACM, New York (2014)
64. Yoshida, T., Hondou, M., Tabata, T., Kan, R., Kiyota, N., Kojima, H., Hosoe, K., Okano, H.: Sparc64 XIfx: Fujitsu's next-generation processor for high-performance computing. IEEE Micro **35**(2), 6–14 (2015). https://doi.org/10.1109/MM.2015.11
65. Young, J.W.: A first order approximation to the optimal checkpoint interval. Commun. ACM **17**(9), 530–531 (1974). http://doi.acm.org/10.1145/361147.361115