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

Carsten Weinhold, Adam Lackorzynski, Jan Bierbaum, Martin Küttler, Maksym Planeta, Hermann Härtig, Amnon Shiloh, Ely Levy, Tal Ben-Nun, Amnon Barak, Thomas Steinke, Thorsten Schütt, Jan Fajerski, Alexander Reinefeld, Matthias Lieber, and Wolfgang E. Nagel

Abstract In this paper we describe the hardware and application-inherent challenges that future exascale systems pose to high-performance computing (HPC) and propose a system architecture that addresses them. This architecture is based on proven building blocks and few principles: (1) a fast light-weight kernel that is supported by a virtualized Linux for tasks that are not performance critical, (2) decentralized load and health management using fault-tolerant gossip-based information dissemination, (3) a maximally-parallel checkpoint store for cheap checkpoint/restart in the presence of frequent component failures, and (4) a runtime that enables applications to interact with the underlying system platform through new interfaces. The paper discusses the vision behind FFMK and the current state of a prototype implementation of the system, which is based on a microkernel and an adapted MPI runtime.

C. Weinhold (✉) • A. Lackorzynski • J. Bierbaum • M. Küttler • M. Planeta • H. Härtig
Department of Computer Science, TU Dresden, Dresden, Germany
e-mail: carsten.weinhold@tu-dresden.de

A. Shiloh • E. Levy • T. Ben-Nun • A. Barak
Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel
e-mail: amnon@cs.huji.ac.il

T. Steinke • T. Schütt • J. Fajerski • A. Reinefeld
Zuse Institute Berlin, Berlin, Germany
e-mail: ar@zib.de

M. Lieber • W.E. Nagel
Center for Information Services and HPC, TU Dresden, Dresden, Germany
e-mail: wolfgang.nagel@tu-dresden.de

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*, Lecture Notes in Computational Science and Engineering 113,
DOI 10.1007/978-3-319-40528-5_18

1 Exascale Challenges

Many reports and research papers, e.g. [12, 14, 19, 25], highlight the role of systems software in future exascale computing systems. It will gain importance in managing dynamic applications on heterogeneous, massively parallel, and unreliable platforms—a burden that cannot be the responsibility of application developers alone anymore, but has to shift to the operating system and runtime (OS/R). The starting point for the design of FFMK is the expectation that these major challenges have to be addressed by systems software for exascale systems:

Dynamic Applications Current high-end HPC systems are tailored towards extremely well-tuned applications. Tuning of these applications often includes significant load balancing efforts [11, 23, 38]. We believe a major part of this effort will have to shift from programmers to OS/Rs because of the complexity and dynamics of future applications. Additionally, exascale applications will need to expose more fine-grained parallelism, leading to new challenges in thread management. A number of runtime systems already addresses these challenges, notably Charm++ [1] and X10 [26]. We further believe that an exascale operating system must accommodate *elastic application partitions* that extend and shrink during their runtime. Still, the commonly used batch schedulers assume fixed size partitioning of hardware resources and networks. FFMK plans to provide interfaces for the cooperation between applications and their runtime to coordinate application-level balancing with overall system management.

Increasing Heterogeneity of Hardware Many current high-end HPC systems consist of compute nodes with at most two types of computing elements, a general purpose CPU (like x86) and an accelerator (like GPGPUs). These elements are assumed and selected to perform very regularly. We assume future hardware will have less regular performance due to fabrication tolerances and thermal concerns. This will add to the unbalanced execution of applications. We also assume that not all compute elements can be active at all time (dark silicon). In addition we assume that other types of computing elements can be expected, for example FPGAs. We believe that systems software can be adapted to such hardware more easily, if the lowest level of software is a small light-weight kernel (LWK) instead of a large and complex system like the Linux kernel.

Higher Fault Rates The sheer size of exascale computers with an unprecedented number of components will have significant impact on the failure-in-time rate for applications. Some OS/Rs already address this concern by enabling incremental and application-specific checkpoint/recovery and by using on-node memory to store checkpoint data. We believe a systems software design for exascale machines must contain a coordinated approach across system layers. For example, runtime checkpointing routines should be able to make use of memory management mechanisms at the OS level to support asynchronous checkpoints.

Deeper Memory Hierarchies We expect more types of memory that differ in aspects like persistence, energy requirements, fault tolerance, and speed. Important examples are on-node non-volatile memory (phase-change memory, flash, etc.) and stacked DRAM. A highly-efficient checkpoint store requires an integrated architecture that makes optimal use of these different types of memory.

Energy Constraints We understand, that provision and running cost of energy will become a—if not the—dominating cost and feasibility factor. To address this problem, we postulate that systems software should be based on an energy model of the complete system. The model should enable a design where each resource management decision can be controlled based on energy/utility functions for resources. For example, an on-node scheduler may choose between running one core at higher speed than others to balance execution times of compute processes. The scheduler's decision should be based on knowledge about which option provides the required cycles at the lowest energy and automatically-inferred predictions of how much time and memory certain computations (e.g., time steps) require.

2 FFMK Architecture Overview

We believe that a systems software design for exascale machines that addresses the challenges described above must be based on a coordinated approach across all layers, including applications. The platform architecture as shown in Fig. 1 uses an L4 microkernel [24] as the light-weight kernel (LWK) that runs on each node.

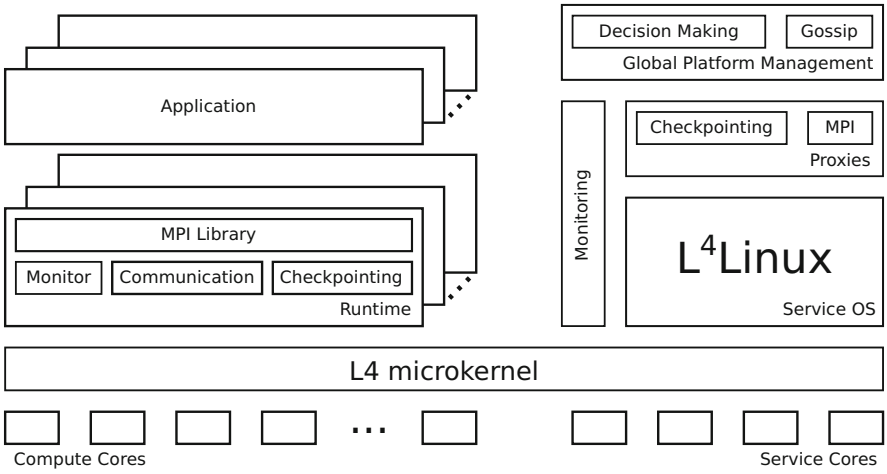

Fig. 1 FFMK software architecture. Compute processes with performance-critical parts of (MPI) runtime and communication driver execute directly on L4 microkernel; non-critical functionality split out into proxy processes on Linux, which also hosts global platform management

All cores are controlled by this minimal common foundation; the microkernel itself is supported by few extra services that provide higher-level OS functionality such as memory management (not shown in the figure). Additionally, an instance of a *service OS* is running on top of it, but only on a few dedicated cores we refer to as "service cores". In our case the system is a full-featured virtualized Linux.

Applications Applications on the system are started by the service OS and can use any functionality offered by it, including device drivers, such as for InfiniBand and network, as well as libraries and programming environments such as MPI. To exercise execution control over the HPC applications, the applications are *decoupled* from the service OS and run independently on the LWK. Any requests of the application to the service OS, such as system calls, are forwarded and handled.

Dynamic Platform Management In the presence of frequent component failures, hardware heterogeneity, and dynamic demands, applications can no longer assume that compute resources are assigned statically. Instead, load and health monitoring is part of the node OS and the platform as a whole is managed by a load distribution service. The necessary monitoring and decision making is done at three levels: (1) on each multi-core node, (2) per application/partition among nodes, and (3) based on a global view of a master management node.

Node-local thread schedulers take care of (1); scalable gossip algorithms disseminate information required to handle (2) and (3). Using gossip, the nodes build up a distributed, inherently fault tolerant, and scalable bulletin board that provides information on the status of the system. Nodes have partial knowledge of the whole system: they know about only a subset of the other nodes, but enough of them in order to make decisions on how to balance load and how to react to failures in a decentralized way. Through new interfaces, applications can pass hints to the local management component, such that it can better predict resource demands and thus help decision making. The global view over all nodes is available to a master node, which receives gossip messages from some nodes. It makes global decisions such as where to put processes of a newly started application.

Fault Tolerance To handle hardware faults, a fast checkpointing module takes intermediate state from applications and distributes and stores it redundantly in various types of memory across several nodes. However, we also envision node-local fault tolerance mechanisms (e.g., replication, micro-reboots) and interfaces to let applications communicate their fault tolerance requirements to the FFMK OS/R.

3 Microkernel-Based Node OS

We have chosen the L4Re microkernel system as basis for node-local OS functionality. For a detailed description of L4, we refer to [24]. In this document, we restrict ourselves to a short intro.

L4 Microkernel L4 had been designed for extensibility rather than as a minimized Unix. As such, it provides few basic abstractions: address spaces, threads, and interprocess communication (IPC). Key ingredient to enabling extensibility is a design that enables both IPC and unblocking of threads to be fast. The IPC mechanism is not only used to transmit ordinary data but also grant access rights to resources, such as capabilities and memory, to other address spaces. On L4, policies are implemented in user-level components. One example is memory management where so-called "pagers" manage the virtual address space of applications and implement any required policy. The microkernel itself only provides the mechanism to grant memory pages.

The fast and simple IPC mechanism enables us to build a componentized FFMK-OS that can achieve high performance. An important feature in this context is that the L4 kernel maps hardware interrupts to IPC messages. As a result, IPC messages can directly wake currently blocked application processes with low latency not only when required input is computed by another process on the same node, but also by processes running on other nodes when messages arrive over the HPC system's interconnect.

Virtualized Linux Our system also runs Linux as a service OS on each node to provide and reuse functionality that is not performance critical such as system initialization. We chose L^{4}Linux, a modified Linux kernel that runs in a virtual machine on the microkernel; it is binary compatible to standard Linux and therefore capable of running unmodified Linux applications.

On the FFMK platform, HPC applications are ordinary Linux programs, too. They are loaded by the service OS and they can use all functionality offered by it, including device drivers and Linux-based runtime environments such as MPI. However, the underlying L4 microkernel is better suited, when applications perform their most "critical" work, which in the context of HPC and exascale systems means "critical to performance". For example, the microkernel can switch context faster than Linux and it provides much better control over what activities run on which core. The latter property is essential to let applications execute undisturbed from the various management and housekeeping tasks that a commodity OS performs in the background.

Decoupled Thread Execution To isolate HPC applications from such "noise", the FFMK OS allows their threads to be *decoupled* from the service OS and run undisturbed on dedicated compute cores. This novel mechanism leverages the tight integration of the paravirtualized commodity kernel and the L4 microkernel. L^{4}Linux uses different L4 address spaces for the Linux kernel and each application process running on top of it. To virtualize CPU cores, it uses a vCPU execution model [20]. Such a vCPU is a special variant of an L4 thread. The Linux scheduler maps all Linux threads to one or more vCPUs, which then migrate between address spaces as they execute either kernel code during Linux system calls or user code of any of the Linux processes. However, since each process on top of L^{4}Linux is backed by its own L4 address space, the code and data contained in it are accessible from all cores in the system, not just those assigned to the service OS.

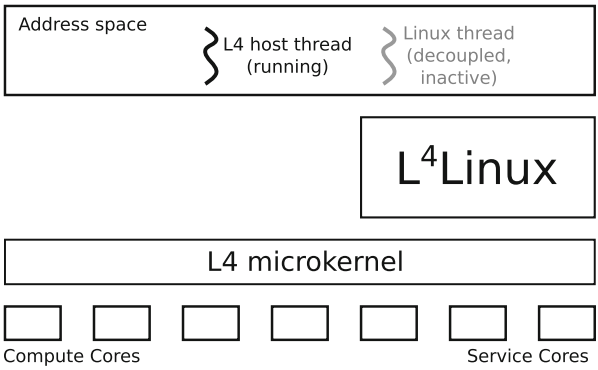

Fig. 2 Split execution model: the paravirtualized L4Linux kernel supports handing off thread execution of Linux programs to the underlying L4 microkernel, such that they can perform computations free of "OS noise" on cores controlled by the L4 microkernel. Decoupled threads are moved back temporarily to a service core assigned to Linux, whenever the program performs a Linux system call

To decouple a thread of a user process from unpredictable Linux behavior, L4Linux creates an additional L4 host thread to execute the application's code. Whenever the application is executing on the host thread, the corresponding Linux thread is detached from the scheduler of the service OS. Since this host thread is put on a separate compute core, which is controlled by L4 directly, it can thus execute in parallel to vCPUs of the service Linux (see Fig. 2). Thus, a noise-sensitive HPC application can run undisturbed and will not be subject to scheduling decisions of Linux, nor will it be interrupted by incoming interrupts.

Decoupled Linux programs can still perform Linux system calls. When doing so, the host thread causes an exception that is forwarded to L4Linux, which then reactivates the decoupled Linux thread and performs the requested operation in its context. Returning from the system call causes the thread to be decoupled again.

Device Access A key advantage of the decoupling mechanism apart from noise reduction is that it fits naturally into high-performance I/O stacks. For example, the InfiniBand driver stack consists of a Linux kernel driver and several user-space libraries (`libibverbs` and `libmlx5` in the case of recent Mellanox InfiniBand cards). These libraries contain the functionality that is on the performance-critical paths, which is why the user-space driver in `libmlx5` has direct access to I/O memory of the host-channel adapter (HCA) without having to call the kernel. Most of the management tasks (e.g., creating queue pairs, registering memory regions) are implemented in the kernel module; the user-space libraries communicate with the in-kernel driver, which is accessible through the system call forwarding as described in the preceding paragraph.

FFMK Node OS The previously described components and mechanisms form the basis of the FFMK node OS. It also hosts a decentralized platform management service which will be described in the next sections.

4 Dynamic Platform Management

FFMK addresses applications with varying resource demands and hardware platforms with variable resource availability (e.g. due to thermal limits or hardware faults). Although the FFMK OS/R is currently limited to node-local scheduling, we envision the full-featured version to dynamically optimize the usage of the application's resources by rebalancing its workload, optimizing network usage, and reacting to changing demands when its *elastic partition* shrinks or expands. Elastic partitions enable the FFMK platform to allocate resources to an application dynamically during the lifetime of the application (see Fig. 3b, c). The main task of the dynamic platform management is to continuously optimize the utilization of the system by means of an economic model. This economic model will include various

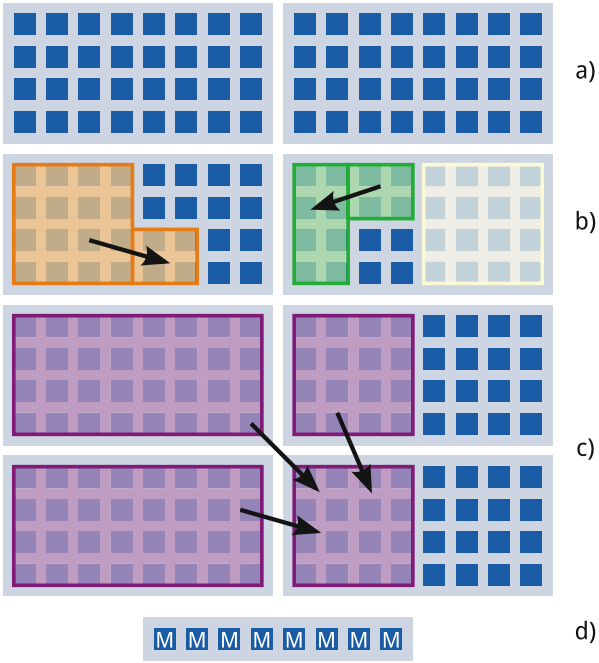

Fig. 3 Dynamic platform management. (**a**) Multicore nodes are organized in colonies. (**b**) Elastic applications partitions can expand and shrink. (**c**) Partitions can span mutliple colonies and expand to new colonies. (**d**) A redundant set of master nodes monitors and controls the system

aspects such as throughput and energy efficiency, fairness among applications, resiliency, and quality of service. However, its details are still subject to research.

The dynamic platform management consists of two basic components: monitoring and decision making. To achieve the scalability and resilience required for exascale systems, we decided to use gossip algorithms for all cross-node information dissemination of the monitoring component (see Sect. 4.2) and make decisions decentralized where possible (see Sect. 4.3).

4.1 Application Model

To support dynamic management of applications on our platform, we require an application model that is more flexible than the coarse-grained and static division of work that common MPI implementations impose. In our model, the decomposition of an application's workload is decoupled from the number execution units. The units of decomposition are migratable *tasks* that communicate with each other (see Fig. 4). For example, a core may run multiple tasks (one after each other) by preempting at blocking communication calls—a principle called overdecomposition [1]. At an abstract level, tasks are units which generate load for different hardware resources (e.g. cores, caches, memory, and network bandwidth) and the OS/R can map them to the hardware in order to optimize the application's performance. There are several reasons why we think this approach makes sense:

- Applications can be decomposed mostly independent from the number of nodes the program uses, which allows sizing the tasks according to the cache size or application-specific data structures.
- If the resource consumption of tasks varies among the tasks and over runtime, the OS/R is able to map and remap tasks intelligently to balance resource usage. This means that the OS/R, and not the application developer, is responsible for load balancing.
- The OS/R shrinks and expands applications to optimize global throughput.
- The OS/R is able to reduce communication costs by doing a communication-aware (re)mapping of tasks to nodes.

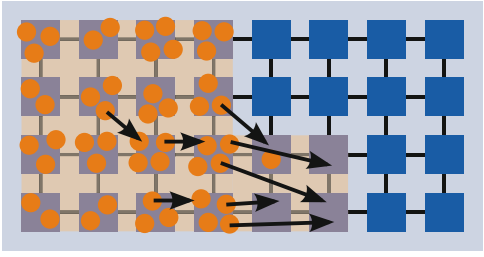

Fig. 4 Applications are decomposed into tasks. Multiple tasks are mapped to a node and can be migrated by the OS/R to expand/shrink the application's partition, to load balance the application, and to optimize communication

- Tasks waiting for a message are not scheduled to a core (i.e. busy waiting is avoided). This allows other tasks to run and to overlap communication with computation. Additionally, the OS/R is able to prioritize tasks that other tasks wait for.
- The OS/R may place tasks of different applications on the same node. Co-locating applications with different resource demands may increase the system utilization and throughput [42].

If, for example, bandwidth is the limiting resource on a node, the OS/R may increase the bandwidth available to the tasks by running fewer of them concurrently and migrating some of the tasks to another node. Additionally, the OS/R may either turn off unneeded cores (to reduce energy consumption) or co-locate bandwidth-insensitive tasks, possibly belonging to another application.

4.2 Monitoring and Gossip-Based Information Dissemination

To be capable of dynamic platform management, the system needs to collect status information about available resources of the nodes and their usage. The status information should contain:

- Current load on the node (cores, caches, memory, memory and network bandwidth)
- Maximum load the node can carry (i.e. available resources, may vary due to faults and thermal limits)
- Communication partners of the tasks running on that node.

The OS/R will use online monitoring (e.g. based on hardware counters) to gather the information on each node. We currently disseminate across node boundaries only information describing the overall resource state of a node. If that turns out to be too coarse-grained, we consider adding information about resource demands of individual tasks. Additionally, applications may pass hints to the runtime that enable a better prediction of future application behavior. The collected and disseminated information is the basis for making decisions as mentioned in the previous section.

Randomized Gossip As briefly introduced in Sect. 2, we will use randomized gossip algorithms to disseminate the resource information and build up the distributed bulletin board. In randomized gossip algorithms each node periodically sends messages with the latest information about other nodes to randomly selected nodes. Received information is merged with the local bulletin board by selecting the newest entry for each node. Thus, each node accumulates local information about the other nodes over time.

We have shown that these algorithms are resilient and they scale to exascale-size systems [5]. Scalability is achieved by dividing the system into *colonies*, each containing in the order of 1000 nodes. The colonies should consist of topologically nearby nodes, see Fig. 3. For the time being we assume that colonies are fixed and

independent of the elastic application partitions. We run the gossip algorithm within each colony independently such that each node knows the status of all other nodes in the same colony; the colonies form the lower level of a gossip hierarchy.

Hierarchical Gossip One level above the colonies, a set of redundant master nodes maintains the global view on all nodes. The masters receive gossip messages from random nodes of each colony to obtain a complete picture of the resource usage and availability of the system. For decentralized decisions concerning multiple colonies (e.g. load balancing of a multi-colony application), the masters additionally send gossip messages with summary information about all colonies back to some colony nodes, which then disseminate it within the colony.

Quality of Information and Overhead Recent results of our research have shown the scalability and resiliency of the randomized gossip algorithms [5]. They work well even when some nodes fail, without the need for any recovery protocol, which is an advantage over tree-based approaches [2]. We developed formal expressions for approximating the average age (i.e., quality of information) of the local information at each node and the information collected by the master. These results closely match the results of simulations and measurements on up to 8192 nodes of a Blue Gene/Q system, as shown in Fig. 5.

We also investigated the overhead of gossip algorithms on the performance of HPC applications sharing network and compute resources [22]. The measurement results for two applications running concurrently to gossip with large information records per node (1024 bytes) are shown in Fig. 6. Sending gossip messages at an interval of 256 ms and above does not cause noticeable overhead, except for extremely communication-intensive codes like MPI-FFT (fast fourier transform).

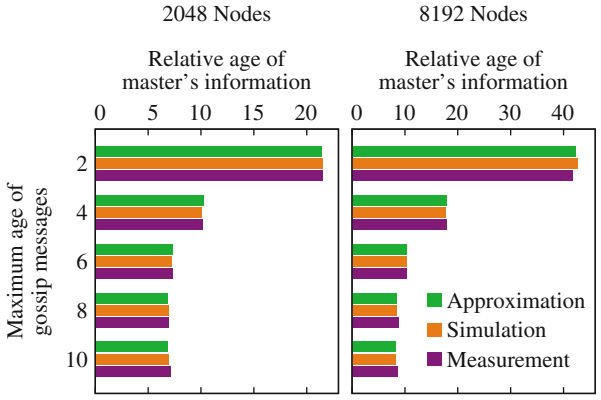

Fig. 5 Average age of the master's information using different age thresholds for gossip message entries (sending only newest information). The age is given relative to the interval of gossip messages. Approximations, simulations, and measurements on Blue Gene/Q match very well

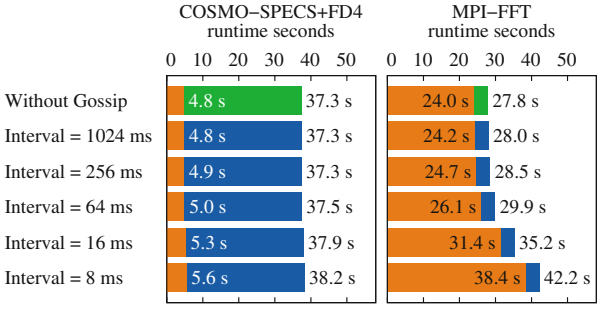

Fig. 6 Runtime overhead of gossip on two benchmark applications on 8192 Blue Gene/Q nodes when varying the interval of gossip messages. The inner red part indicates the MPI portion

4.3 Decision Making

Deciding on how to optimize system utilization is performed at three levels: within each node, decentralized between nodes for each application partition, and centralized at the master nodes. Each level is responsible for a part of the dynamic management of applications as outlined in Sect. 4.1. In the following, we explain the three levels top–down.

- **Whole system:** the master nodes optimize elastic partitions (i.e., shrinking and expanding them), multi-application resource assignment, placement of new partitions, and handling of failures. The master assigns nodes to partitions, but does not care about the mapping of individual tasks to nodes.
- **Per application partition:** gossiping nodes perform decentralized load balancing and communication optimization by migrating tasks within the partition. We will focus on scalable, distributed algorithms that act on small node neighborships or pairs of nodes. Depending on the application behavior, different algorithms will be considered (e.g., Diffusion-based [13], MOSIX [3]). Additionally, partition optimization decisions from the master are realized on the task level, e.g. decide which tasks to migrate to new nodes of an expanded partition.
- **Within each node:** the scheduler of the node OS assigns tasks to cores, taking into account data dependencies and arrival of messages from the network. It also performs dynamic frequency scaling and decides on which execution units to power up (dark silicon).

The FFMK OS makes load management decisions using local knowledge that each node acquired through monitoring and gossip-based information dissemination as described in Sect. 4.2. This information is always about the past, which is not always a good forecast of future behavior of highly dynamic workloads. Therefore, we plan to use techniques to predict resource consumption, like those employed by ATLAS [33]. ATLAS is an auto-training look-ahead scheduler that correlates observed behavior (e.g., execution times, cache misses) and application-provided

information ("metrics") about the next chunk of work to be executed. Applications pass these metrics to the OS to help it make more accurate predictions of future behavior. If, for example, an HPC application's workload in the next time step depends on the number of particles in a grid cell, then this metric (the number of particles) can be used by ATLAS to predict the required compute time to complete the time step; it does so by inferring this information from observed correlation of previous (metric, execution time) pairs. We expect—and hear from application developers—that providing such metrics can be done with little effort. Additionally, applications may inform the OS/R about future workload changes, such that the platform management is able to proactively adapt resource allocations.

5 MPI Runtime

The FFMK architecture is designed such that it can support different runtimes on the LWK at the same time, such as MPI, X10 or Charm++. Due to limited resources and because MPI is the foundation of the vast majority of applications, we focus on dynamizing this traditional HPC runtime such that the FFMK OS can perform load balancing at the OS/R level.

5.1 MPI and Load Balancing

Load balancing applications for exascale HPC systems is a major challenge [14, 25]. For example, in the case of MPI-based applications, each of the participating MPI processes is usually mapped to its own core. If a few MPI processes reach a synchronization point later than the others, the majority of cores become effectively idle, thereby wasting resources. Unfortunately, load imbalances are typical for many important classes of applications, including atmospheric simulations [41], particle simulations [38], and fluid dynamics [17].

Load Balancing by Overdecomposition As explained in the previous sections, the common approach for tackling these load balancing issues is to (1) *overdecompose* by splitting the problem into more parts (i.e., tasks) than cores available, (2) assign the parts to cores, and (3) adapt this mapping dynamically during runtime so as to minimize both imbalance and communication costs. Typically, this method of dynamic load balancing is implemented at the application and library level [23, 38], because MPI implementations do not provide any built-in load management mechanism. This means that the mapping of MPI processes to cores remains static and the application itself is responsible for redistributing workload among ranks to maintain the balance. Even though this approach proved very effective in reducing imbalances and thereby improving performance, it is most often tailored to a specific application or problem domain and cannot be applied to arbitrary workloads easily. Thus, developers are forced to "reinvent the wheel" over and over again.

Adaptive MPI (AMPI) To save developer effort, one could overdecompose at the level of MPI ranks by just creating more ranks than cores available. AMPI [1] is an example of an MPI implementation that does exactly this. It is based on Charm++ [18] and maps each MPI rank to a "chare", which is the Charm++ equivalent of a task. This approach enables the underlying Charm++ runtime system to perform load balancing and migration of MPI ranks transparently. However, chares are not OS-level processes, but C++ objects encapsulating all code and data. Thus, MPI ranks in AMPI share the same address space of a single Charm++ runtime process on each node. Therefore, most MPI applications have to be modified to work on top of AMPI, because global variables are disallowed. Also, multithreaded MPI ranks cannot be supported, because chares are single threaded entities.

5.2 OS/R Support for Oversubscription

Adaptive MPI's compatibility limitations can be overcome by actually creating more MPI compute processes—and thereby more threads—which are subject to a system-level load balancer.

Requirements Analysis The advantage of MPI overdecomposition is that it enables automatic load balancing for MPI applications without having to modify their code. However, it comes at the cost of additional management and communication overhead due to the increased number of ranks. Furthermore, current MPI implementations cause any process that waits for a message transfer to complete to occupy a core, because polling is used. Such busy waiting causes unacceptable overhead in combination with oversubscription, because it effectively prevents overlapping computation and communication. In order for process-level oversubscription to work, waiting must be performed in a blocking fashion instead and the additional overhead must be kept at a minimum to allow for real performance gains. Thus, the OS/R has to provide light-weight message and thread management that allows for fast unblocking of a rank once a message for this rank arrives. Ideally, the system also takes communication dependencies into account when making scheduling decisions: it should prioritize those communication partners that other processes are waiting for so as to keep message latency low.

Preliminary Study To assess the potential of this approach, we conducted a preliminary study where we used MVAPICH2 [29] for oversubscribed runs of the weather simulation code COSMO-SPECS+FD4 [23] and the atomistic simulation CP2K [30]. Both are prone to load imbalances.[1] We used a small FDR InfiniBand test cluster with four nodes that ran a standard GNU/Linux system, since Linux

[1] COSMO-SPECS+FD4 has an internal load balancer, which we disabled in the experiments described here.

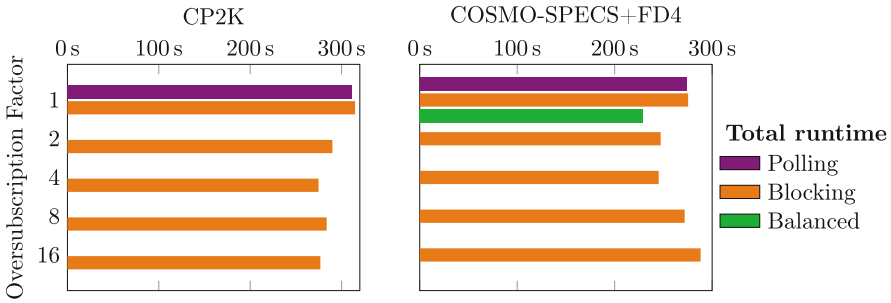

Fig. 7 Preliminary oversubscription study with the applications CP2K and COSMO-SPECS+FD4 using MVAPICH2 on a 16-core/4-node InfiniBand test cluster

kernels preinstalled on HPC systems are typically tuned to not migrate processes between cores. MVAPICH2 does not only support native InfiniBand as a communication back-end but also allows for blocking communication, where the library will block in the kernel until a new message arrives instead of actively polling for messages.

We found that blocking causes only a small overhead compared to busy waiting, as shown in Fig. 7 for the two applications: the purple bars show the runtime when using polling (traditional MPI behavior), the orange bars below show the same benchmark with blocking enabled. However, the results also indicate that overdecomposition and oversubscription of MPI processes can indeed improve performance. Compared to the configurations at the top of the diagrams, which show the total runtime with one MPI process per core (i.e., oversubscription factor of 1), we can see significant improvements in those runs where we oversubscribed the cores by a factor of up to 16 times. The workload remained the same in all cases; we just increased the number of MPI ranks working on the problem.

The MPI library was configured to block in the kernel when waiting for messages; no busy waiting was performed in MPI routines. This allows the scheduler of the Linux OS to migrate threads among cores in order to utilize all cores equally, thereby overlapping wait times with computations in other MPI processes.

For comparison, we also give the runtime of COSMO-SPECS+FD4 with its internal load balancer enabled (green bar labeled "balanced"). We can see that OS-level oversubscription still does not achieve the same performance, but it gets within 7 % at 4× oversubscription. The improvement in the oversubscribed configuration is achieved with no effort from the developer's side; in contrast, several person years went into COSMO-SPECS+FD4's load balancer.

More results of oversubscription experiments, also showing the benefit of multiple applications sharing the same nodes, are described in a tech report [37].

6 Migration

The FFMK prototype does not support inter-node process migration yet. It can only balance load within each node, where the OS scheduler migrates threads among cores. Nevertheless, we regard migration as the "swiss army knife" of an exascale OS/R: this mechanism can be used to (1) further improve load balancing, for (2) proactive fault tolerance as described in Sect. 7, and (3) as a tool for achieving better energy efficiency.

The Case for Migration Migration of MPI processes within a single node is taken care of by the local scheduler of the node OS. However, this approach to load balancing is no longer optimal, if the total amount of work per node varies within the application partition (i.e., the processes on some nodes take longer than on others). An example of this situation is shown for CP2K in Fig. 8. It visualizes how much time each of the 1024 MPI ranks spent doing useful computation in each time step. Green indicates a high computation/communication ratio, whereas yellow and red areas of the heatmap show that most of the time is spent in MPI waiting for communication to finish.

To reduce the load imbalance, nodes hosting "green" processes need to migrate some MPI ranks to nodes that are mostly red and yellow. Fortunately, our analysis of CP2K and other applications such as COSMO-SPECS+FD4 revealed that the

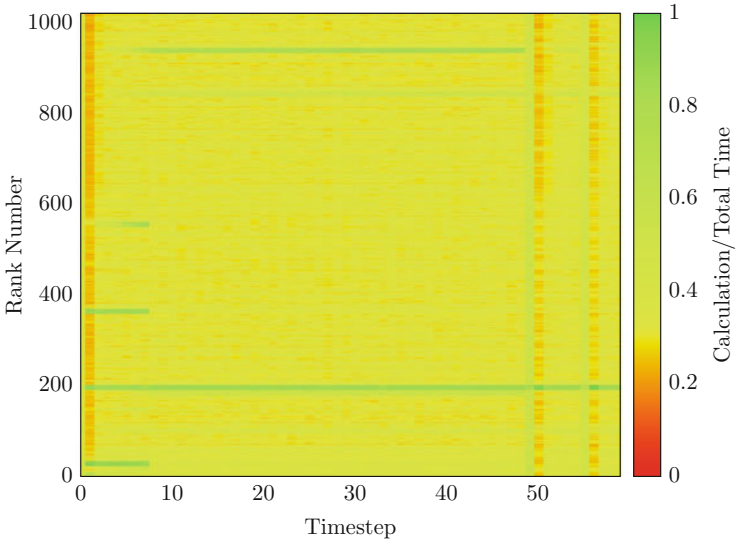

Fig. 8 Load imbalances in CP2K. Colors show computation vs communication ratio of each MPI process *(Y axis)* per time step *(X axis)*. *Yellow* and *red* indicate short computation time vs long waiting for other MPI ranks; a small number of overloaded processes delay all others, because they need significantly longer to compute their chunk of work in a time step *(green areas)*

load caused by each process changes rather slowly, if at all. This observation is encouraging, because inter-node migration takes much more time than migrating a thread within the same node, but can be performed less frequently.

Migration Obstacles Inter-node migration is complicated due to the static nature of communication back-ends such as InfiniBand and MPI itself. For the benefit of performance, implementations are designed such that after an initial setup phase, modifications to the partners involved in a communication are not easily possible. The RDMA-based job migration scheme [31] by Ouyang et. al. addresses this problem by tearing down all communication endpoints prior to migration and re-establishes them when the application resumes. The approach [4] taken by Barak et al. only works with TCP/IP-based communication. Despite these research efforts and others in the area [36], migration has never been integrated into production MPI libraries, even though the MPI standard [28] does not prohibit this feature.

Checkpoint-Migrate-Restart Given that transparent inter-node migration is hard with state-of-the-art communication stacks, and since it is needed only infrequently, we consider a simpler solution to the load-balancing problem that is based on coordinated checkpoint/restart (C/R): to migrate individual MPI processes, we (1) checkpoint at a convenient time (e.g., after completing a time step) the current state of the whole application, (2) terminate all processes, and then (3) restart them, but with certain processes assigned to previously underloaded nodes. The new placement of "migrated" processes is determined based on system monitoring and decision making as described in Sects. 4.2 and 4.3, respectively.

Checkpoint/Restart Approach The efficacy of the approach relies on the ability of the system to perform checkpoint/restart with very low overhead. A key metric to optimize is the amount of data that needs to be checkpointed and/or sent over the network. Compared to system-level C/R solutions such as BLCR [8], application-assisted checkpointing usually produces much smaller state. The reason is that they serialize just the internal state that is needed to restart, but not the contents of entire address spaces. Application-specific C/R support is common in HPC codes. There are also frameworks such as SCR [27] that support multi-level checkpointing, where data is stored in memory before it is transferred to persistent storage in the background. On the other hand, support for BLCR-like system-level solutions has been deprecated recently, or removed entirely from major MPI implementations. We therefore focus on application-assisted checkpoint/restart as the process-migration mechanism in the FFMK OS, but system-level C/R would work, too.

Furthermore, earlier work by Ouyang et al. [31] found that the restart phase takes by far the longest time in this migration scheme. We can confirm that re-initialization after restarting is still a major factor, but also one that leaves room for optimizations. For example, we found that, in MVAPICH2, `MPI_Init` spends several hundred milliseconds to obtain topology information about the local node using the hwloc library. Older versions of the MPI library also called initialization routines of the InfiniBand driver stack multiple time. This overhead

can be eliminated by caching results or removing any redundant function calls; we submitted patches that fix the latter performance issue to the MVAPICH2 authors.

Finally, to achieve the level of performance for C/R to be usable as a migration mechanism, we employ in-memory checkpointing to make serialized application state accessible from any node where processes are migrated to. The next section on fault tolerance techniques covers requirements for a suitable checkpoint store.

7 Fault Tolerance

The HPC research community expects that the total number of individual components in exascale machines will grow dramatically. It is already becoming increasingly common to add more levels of node-local memory (e.g., SSDs), and heterogeneous architectures using accelerators are state of the art. This increased complexity and the expectation of higher failure rates for individual components and the whole system require a much more sophisticated approach to fault tolerance. In the following paragraphs, we give an overview of the key techniques and how they fit into the FFMK architecture.

Protecting Applications: Checkpoint/Restart The state-of-the-art mechanism to protect applications from crashes and other fail-stop errors is to make their execution state recoverable using checkpoint/restart (C/R) [10, 34]. FFMK aims at integrating a high-performance C/R system that utilizes the distributed storage built into all nodes of an exascale system, instead of relying on a traditional parallel file system that is connected to the supercomputer via a small number of I/O nodes. The general approach has been shown to scale extremely well with the number of nodes, for example in work by Rajachandrasekar et al. [32].

The FFMK project implements scalable C/R based on XtreemFS [40]. Due to space constraints, we do not discuss this distributed file system in detail, but give only a brief summary: XtreemFS supports storing erasure-coded file contents (e.g., checkpointed application state) in local memory of (potentially all) nodes of an HPC system. Erasure coding ensures that data is still accessible even if multiple nodes fail; at the same time, it minimizes both the network bandwidth required to transmit checkpoint data over the network and the amount of on-node storage that is required.

Proactive Fault Handling The FFMK OS' automatic load management and migration support (see Sect. 6) can also be used for proactive fault tolerance similar to [36]. By migrating all processes away from a node that is about to fail, the system can keep applications running without having to restart them from a checkpoint. To this end, FFMK leverages the hardware monitoring and information dissemination support described in Sect. 4.2: if a node observes critical CPU temperatures or correctable bit flips in a failing memory bank, it can initiate migration of all local processes to another node. We also consider partial node failures, where, for example, a single core becomes unreliable, but all other cores continue working properly. In both cases, the system may temporarily oversubscribe

healthy resources (other nodes or unaffected local cores) by migrating processes. We consider any slowdowns caused by such "emergency evacuations" a special case of load imbalance, which can be resolved either by the FFMK load balancing system or by assigning replacement nodes to the application.

Resilient Gossip Algorithms At the system level, however, the FFMK OS relies on fault-tolerant algorithms. The most important ones are the randomized gossip algorithms, which are used to propagate information about the health of each node. Furthermore, they indirectly allow the system to identify nodes that stopped responding (e.g., due to an unexpected crash or network failures). The algorithms themselves are inherently fault-tolerant and they provide good quality of information even when some of the participating nodes failed; details of the theoretical foundations and simulation are discussed in [5].

The overview on fault tolerance concludes the presentation of the FFMK architecture. In the next section, we discuss related work.

8 Related Work

There exist several other projects that build operating systems for future HPC systems. In the following, we will characterize the projects from our point of view and emphasize the differences.

Argo and Hobbes The first two OSes, Hobbes [9] and Argo [6], are based on a general architecture similar to ours. They include a node OS as basis, global platform management, and an intermediate runtime providing a light weight thread abstraction. To our knowledge, the global management in both cases is based on MRNet [2], a fault-tolerant tree management structure, whereas FFMK uses gossip algorithms [5] for their inherent fault tolerance properties. The Argo consortium includes the research group behind Charm++ [18] to provide a versatile load balancing and resource management together with a light weight thread-like abstraction. The philosophy behind Charm++ is similar to our task-based application model. Argo uses Linux as the basis of their node OS. Hobbes is based on a newly built microkernel named Kitten [21]. In contrast to L4, Kitten's interface resembles the Unix interface, but is cut down and tailored towards enabling Linux applications to run directly on the microkernel. As does FFMK, Hobbes also relies on virtualization technology to support Linux applications that require features not provided by the microkernel; system calls not supported by Kitten are forwarded to Linux.

mOS The mOS project [39] at Intel is also based on a light-weight kernel (LWK) that runs colocated with a fully-fledged Linux. System calls that are not supported by the LWK are forwarded to the Linux kernel. However, in contrast to the FFMK platform, the mOS LWK controls only compute cores, whereas the L4 microkernel of our OS platform is in control of all cores.

Manycore OS Riken's OS [35] developed under Yukata Ishikawa also is a hybrid system. To the best of our knowledge, the main difference compared to FFMK is the fact that the microkernel can run on accelerators such as Xeon Phi, but remains under control of a Linux. The system pioneered splitting the InfiniBand driver stack, such that processes running on the accelerators can reuse the functionality hosted on Linux by way of communication between Linux and the microkernel.

9 Summary and Future Work

State of the Union In this paper, we described the challenges that future HPC systems pose to system and application developers. Based on these challenges, we motivated an architecture for an exascale operating system and runtime (OS/R): the microkernel-based FFMK OS. We described the current state of our prototype implementation, which, at the time of this writing, is capable of running unmodified MPI applications. The implementation of the node OS consists of an L4 microkernel, which is supported by a virtualized Linux kernel that we use as a service OS. While our gossip algorithms are well-studied and found to be suitable, the decision making algorithms that build on top are not yet implemented; gobal platform management is therefore not part of the prototype. However, the node OS has been successfully tested on a 112-node InfiniBand cluster across 1,344 Intel Xeon cores.

Future Work Our short-term agenda focuses on evaluating process-level overdecomposition and oversubscription of MPI applications (see Sect. 5). Furthermore, our work on the "decoupled thread" execution model presented in Sect. 3 is currently under peer review. The FFMK project is funded for three more years, during which we plan to finalize and integrate those building blocks of the architecture that are not yet complete. This includes especially the checkpoint/restart layer and cross-node migration support.

A key area of future work in the long term is research into novel interfaces between applications and the OS/R. We already have experience with schedulers [33] that can make better decisions based on application-provided hints about future behavior. We also investigated "programming hints" for optimizing memory accesses in GPU-based applications [7]. Application-level hints seem also promising for fault tolerance: HPC application developers [15] are already researching fault-tolerant versions of the core algorithms used in their HPC codes. Such codes may be able to handle node failures without restarting from a checkpoint, provided that the application can inform the OS/R about its fault tolerance requirements through a suitable interface.

Acknowledgements 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 [16]. 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 supercomputer.

References

1. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., Kale, L.: Parallel programming with migratable objects: Charm++ in practice. In: Proceedings of the Supercomputing 2014, Leipzig, pp. 647–658. IEEE (2014)
2. Arnold, D.C., Miller, B.P.: Scalable failure recovery for high-performance data aggregation. In: Proceedings of the IPDPS 2010, Atlanta, pp. 1–11. IEEE (2010)
3. Barak, A., Guday, S., Wheeler, R.: The MOSIX Distributed Operating System: Load Balancing for UNIX. Lecture Notes in Computer Science, vol. 672. Springer, Berlin/New York (1993)
4. Barak, A., Margolin, A., Shiloh, A.: Automatic resource-centric process migration for MPI. In: Proceedings of the EuroMPI 2012. Lecture Notes in Computer Science, vol. 7490, pp. 163–172. Springer, Berlin/New York (2012)
5. Barak, A., Drezner, Z., Levy, E., Lieber, M., Shiloh, A.: Resilient gossip algorithms for collecting online management information in exascale clusters. Concurr. Comput. Pract. Exper. **27**(17), 4797–4818 (2015)
6. Beckman, P., et al.: Argo: an exascale operating system. http://www.argo-osr.org/. Accessed 20 Nov 2015
7. Ben-Nun, T., Levy, E., Barak, A., Rubin, E.: Memory access patterns: the missing piece of the multi-GPU puzzle. In: Proceedings of the Supercomputing 2015, Newport Beach, pp. 19:1–19:12. ACM (2015)
8. Berkeley Lab Checkpoint/Restart. http://ftg.lbl.gov/checkpoint. Accessed 20 Nov 2015
9. Brightwell, R., Oldfield, R., Maccabe, A.B., Bernholdt, D.E.: Hobbes: composition and virtualization as the foundations of an extreme-scale OS/R. In: Proceedings of the ROSS'13, pp. 2:1–2:8. ACM (2013)
10. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Automated application-level checkpointing of MPI programs. ACM Sigplan Not. **38**(10), 84–94 (2003)
11. Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T., Wilcox, L.: Extreme-scale AMR. In: Proceedings of the Supercomputing 2010, Tsukuba, pp. 1–12. ACM (2010)
12. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innov. **1**(1), 5–28 (2014)
13. Corradi, A., Leonardi, L., Zambonelli, F.: Diffusive load-balancing policies for dynamic applications. IEEE Concurr. **7**(1), 22–31 (1999)
14. Dongarra, J., et al.: The international exascale software project roadmap. Int. J. High Speed Comput. **25**(1), 3–60 (2011)
15. EXAHD – An Exa-Scalable Two-Level Sparse Grid Approach for Higher-Dimensional Problems in Plasma Physics and Beyond. http://ipvs.informatik.uni-stuttgart.de/SGS/EXAHD/index.php. Accessed 29 Nov 2015
16. FFMK Website. http://ffmk.tudos.org. Accessed 20 Nov 2015
17. Harlacher, D.F., Klimach, H., Roller, S., Siebert, C., Wolf, F.: Dynamic load balancing for unstructured meshes on space-filling curves. In: Proceedings of the IPDPSW 2012, pp. 1661–1669. IEEE (2012)

18. Kale, L.V., Zheng, G.: Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: Parashar, M., Li, X. (eds.) Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications, chap. 13, pp. 265–282. Wiley, Hoboken (2009)
19. Kogge, P., Shalf, J.: Exascale computing trends: adjusting to the "New Normal" for computer architecture. Comput. Sci. Eng. **15**(6), 16–26 (2013)
20. Lackorzynski, A., Warg, A., Peter, M.: Generic virtualization with virtual processors. In: Proceedings of the 12th Real-Time Linux Workshop, Nairobi (2010)
21. Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., Brightwell, R.: Palacios and Kitten: new high performance operating systems for scalable virtualized and native supercomputing. In: Proceedings of the IPDPS 2010, Atlanta, pp. 1–12. IEEE (2010)
22. 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 the ROSS'14, Munich, pp. 10:1–10:7. ACM (2014)
23. 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: Proceedings of the PARA 2010. Lecture Notes in Computer Science, vol. 7133, pp. 131–141. Springer, Berlin/New York (2012)
24. Liedtke, J.: On micro-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95), Copper Mountain Resort, pp. 237–250. ACM (1995)
25. Lucas, R., et al.: Top ten exascale research challenges. DOE ASCAC subcommittee report. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf (2014). Accessed 20 Nov 2015
26. Milthorpe, J., Ganesh, V., Rendell, A.P., Grove, D.: X10 as a parallel language for scientific computation: practice and experience. In: Proceedings of the IPDPS 2011, Anchorage, pp. 1080–1088. IEEE (2011)
27. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.: Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system. Technical report LLNL-TR-440491, Lawrence Livermore National Laboratory (LLNL) (2010)
28. MPI: A message-passing interface standard, version 3.1. http://www.mpi-forum.org/docs (2015). Accessed 20 Nov 2015
29. Mvapich: Mpi over infiniband. http://mvapich.cse.ohio-state.edu/. Accessed 20 Nov 2015
30. Open Source Molecular Dynamics. http://www.cp2k.org/. Accessed 20 Nov 2015
31. Ouyang, X., Marcarelli, S., Rajachandrasekar, R., Panda, D.K.: RDMA-based job migration framework for MPI over Infiniband. In: Proceedings of the IEEE CLUSTER 2010, Heraklion, pp. 116–125. IEEE (2010)
32. Rajachandrasekar, R., Moody, A., Mohror, K., Panda, D.K.: A 1 PB/s file system to checkpoint three million MPI tasks. In: Proceedings of the HPDC'13, New York, pp. 143–154. ACM (2013)
33. Roitzsch, M., Wachtler, S., Härtig, H.: Atlas: look-ahead scheduling using workload metrics. In: Proceedings of the RTAS 2013, Philadelphia, pp. 1–10. IEEE (2013)
34. Sato, K., Maruyama, N., Mohror, K., Moody, A., Gamblin, T., de Supinski, B.R., Matsuoka, S.: Design and modeling of a non-blocking checkpointing system. In: Proceedings of the Supercomputing 2012, Venice, pp. 19:1–19:10. IEEE (2012)
35. Sato, M., Fukazawa, G., Yoshinaga, K., Tsujita, Y., Hori, A., Namiki, M.: A hybrid operating system for a computing node with multi-core and many-core processors. Int. J. Adv. Comput. Sci. **3**, 368–377 (2013)
36. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration and back migration in HPC environments. J. Par. Distrib. Comput. **72**(2), 254–267 (2012)
37. Wende, F., Steinke, T., Reinefeld, A.: The impact of process placement and oversubscription on application performance: a case study for exascale computing. Technical report 15–05, ZIB (2015)

38. Winkel, M., Speck, R., Hübner, H., Arnold, L., Krause, R., Gibbon, P.: A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations. Comput. Phys. Commun. **183**(4), 880–889 (2012)
39. Wisniewski, R.W., Inglett, T., Keppel, P., Murty, R., Riesen, R.: mOS: an architecture for extreme-scale operating systems. In: Proceedings of the ROSS'14, Munich, pp. 2:1–2:8. ACM (2014)
40. XtreemFS – a cloud file system. http://www.xtreemfs.org. Accessed 20 Nov 2015
41. Xue, M., Droegemeier, K.K., Weber, D.: Numerical prediction of high-impact local weather: a driver for petascale computing. In: Bader, D.A. (ed.) Petascale Computing: Algorithms and Applications, pp. 103–124. Chapman & Hall/CRC, Boca Raton (2008)
42. Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., Klasky, S.: Goldrush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In: Proceedings of the Supercomputing 2013, Eugene, pp. 78:1–78:12. ACM (2013)