FOX: A Fault-Oblivious Extreme-Scale Execution Environment Final Report

FOX team

LLNL, Sandia CA, PNNL, Boston U., Ohio State U., IBM, Bell Labs

Executive Summary

The FOX project, funded under the ASCR Xstack I program, developed systems software and runtime libraries for a new approach to the data and work distribution for massively parallel, fault oblivious application execution. Our work was motivated by the premise that exascale computing systems will provide a thousand-fold increase in parallelism and a proportional increase in failure rate relative to today's machines. To deliver the capability of exascale hardware, the systems software must provide the infrastructure to support existing applications while simultaneously enabling efficient execution of new programming models that naturally express dynamic, adaptive, irregular computation; coupled simulations; and massive data analysis in a highly unreliable hardware environment with billions of threads of execution.

Our OS research has prototyped new methods to provide efficient resource sharing, synchronization, and protection in a many-core compute node. We have experimented with alternative task/dataflow programming models and shown scalability in some cases to hundreds of thousands of cores. Much of our software is in active development through open source projects. Concepts from FOX are being pursued in next generation exascale operating systems.

Our OS work focused on adaptive, application tailored OS services optimized for multi $\rightarrow$ many core processors. We developed a new operating system NIX that supports role-based allocation of cores to processes which was released to open source. We contributed to the IBM FusedOS project, which promoted the concept of latency-optimized and throughput-optimized cores. We built a task queue library based on distributed, fault tolerant key-value store and identified scaling issues. A second fault tolerant task parallel library was developed, based on the Linda tuple space model, that used low level interconnect primitives for optimized communication. We designed fault tolerance mechanisms for task parallel computations employing work stealing for load balancing that scaled to the largest existing supercomputers. Finally, we implemented the Elastic Building Blocks runtime, a library to manage object-oriented distributed software components. To support the research, we won two INCITE awards for time on Intrepid (BG/P) and Mira (BG/Q).

Much of our work has had impact in the OS and runtime community through the ASCR Exascale OS/R workshop and report, leading to the research agenda of the Exascale OS/R program. Our project was, however, also affected by attrition of multiple PIs. While the PIs continued to participate and offer guidance as time permitted, losing these key individuals was unfortunate both for the project and for the DOE HPC community.

FOX PIs The FOX team encompassed seven institutions (three National Labs, two universities, and two industrial partners. PIs were Ronald Minnich, Curt Janssen, Jeremiah J. Wilke and John Floren (Sandia CA), Maya Gokhale, Roger Pearce, Scott Lloyd (Lawrence Livermore National Laboratory), Sriram Krishnamoorthy (Pacific Northwest National Laboratory), Jonathan Appavoo (Boston University), P. Sadayappan (Ohio State University), Eric Van Hensbergen, Evan Speight, Jimi Xenedis (IBM Research), Jim McKie and Noah Evans (Bell Labs). In the course of the project, Ron Minnich and Curt Janssen went to Google, and Eric Van Hensbergen went to ARM.

1 Introduction

As the High Performance Computing community prepares for the extreme scale of execution to be provided by exascale computing systems, many HPC experts [1] believe that far reaching changes in hardware required for the exascale will require corresponding innovation in system software, runtime libraries, and applications.

Each new generation of HPC systems has presented difficult but manageable challenges. Power for tera- and petascale systems was managed by adding more money to the power budget; scalable performance involved using an existing tool set to measure and modify existing libraries so that existing applications could directly drive the network. The envisioned reliability problems[1] were resolved with careful design and fabrication, such that even petascale systems stay up for many days. Even the perceived need for a custom kernel turned out to be wrong, as Linux can be used off-the-shelf, when built with enough configuration tweaks.

However, exascale systems will be hitting several walls at the same time. Nodes will have many cores, and each node will run many different processes in support of a single application. These processes will compete for memory, network, and power resources, and must be kept from interfering with one another by running in an unprivileged mode. Power management will be highly dynamic, no longer simply powering up all the nodes at full clock rate. It is expected that due to scale, failure will be more the norm than the exception. As a consequence, the operating system will have a much larger role to play in memory protection, resource allocation, power management, and resilience. For applications to actually benefit from the hundred-fold increase in parallelism relative to today's multi-petaflop machines the community must create new capabilities in system software infrastructure, supporting the evolution from single process per node "MPI+X" to new programming models that naturally express the application's dynamic, adaptive, irregular execution patterns (e.g. Figure 1) in an unreliable hardware environment with billions of threads of execution.

In this project, we have developed system software and runtime support for the massively parallel, dynamic application execution that we expect at the exascale. Figure 2 illustrates a many-core node and our system software stack. The hybrid runtime consists of a general purpose Service OS along with a light-weight library OS (if needed). A variety of runtime libraries support a wide granularity of parallel tasks within a node and within collections of nodes. We report on our experience in prototyping and evaluating performance and resilience of these advanced operating systems and runtime mechanisms from single node level experiments up to petaflop systems. Most of the software discussed is available as open source, and some components are being incorporated into next generation experimental OS projects.

2 Operating systems

Our operating system design targets a node architecture comprising many, possibly heterogenous cores. We optimize for a hardware environment in which cores have different roles, as exemplified by (perhaps) different instruction set architectures, different clock frequencies, and differing ability to run supervisor level code.

An implication of this change of node architecture is that exascale systems are expected to run multiple processes, not just at the whole machine level, but at the node or socket level. Some of the processes might perform different parts of the computation, while others might monitor and steer it. The effects of this

[1] The LANL Q, a 30T machine, was projected to have a failure every 20 minutes. The projections were off by a factor of at least 100.

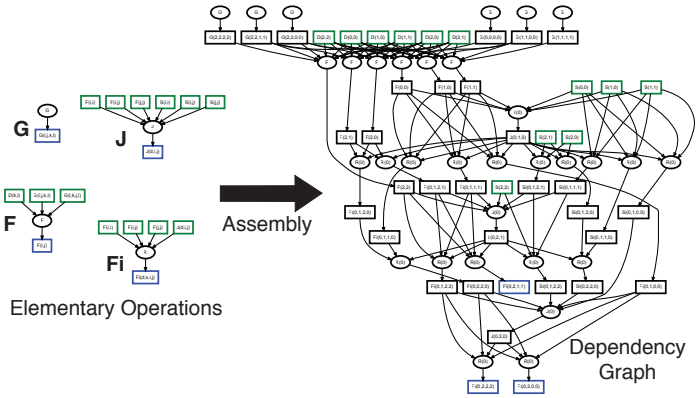

Figure 1: Dynamic dependence graph of a Hartree-Fock computation

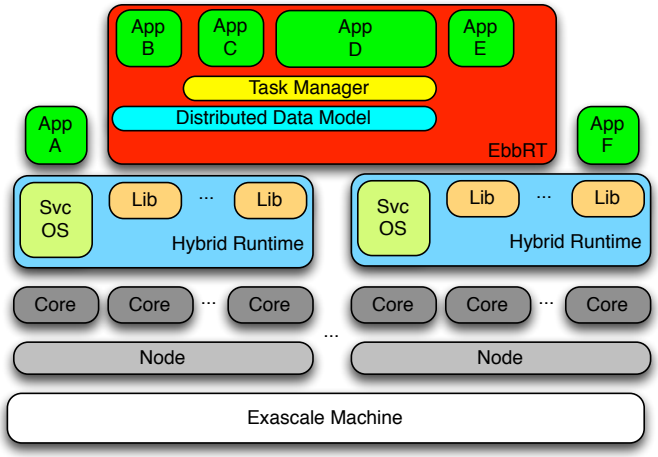

Figure 2: Operating system and runtime stack for extreme scale execution

change are hard to overstate. Most production HPC applications run in SPMD mode. Instances of the same program run as single processes, one per node, and each user process has direct access to memory and I/O devices. Many lightweight kernels for HPC systems cannot support multiple processes or enforce standard memory protections. Network interface code is managed in libraries, not in the kernel, to achieve optimum performance.

Our work on HPC operating systems anticipates the change to multiple, heterogeneous processes that need OS services. We consider the operating system an important component of the exascale software stack, rather than something to be shunted out of the way once the application has started. Application I/O is managed by the operating system rather than by OS bypass. The decision to keep the operating system involved in I/O has performance impact, and we describe below some of the ways we overcome those issues. In the course of our research, we developed two OS variants based on the Plan 9 kernel [2], HARE and NIX. In the HARE OS [3] on the Blue Gene/P (Section 2.1), we developed novel OS techniques to optimize OS servicing of user process communication. In the NIX kernel [4], prototyped on x86 nodes (Section 2.2), we designed and implemented core specialization in the OS.

2.1 Improving operating system efficiency

Our goal was to replace OS bypass (in which an application interacts directly with a device such as network interface) with kernel-based mechanisms. OS bypass was implemented to provide a way to avoid the performance impact of OS involvement in I/O, especially the latency the OS adds for small messages. Small message performance is crucial to the global performance of applications, since small messages are used in synchronization, barriers, and other time-critical operations.

We focused on small message performance first. The first problem was to be able to send them with very low latency; the second was how to respond to them with very low overhead.

Reducing Send Overhead with Customized, Per-Process System Calls. For the small messages used in barriers and reductions, we found that in practice only a small number of different parameter values are used. We provided a mechanism for the application to optimize device I/O by creating a new system call in which constant parameters are eliminated (a form of *currying*): Given a system call with constant parameters, a process can create its own private system call, with the parameters pre-computed. The kernel has a fast path for recognizing these private system calls, which connects to a fast path in the driver. The new system call is private to the process and its children. Validation of system call parameters is done when the call is created, instead of each time. On Blue Gene/P, we were able to get the data to the wire in 700 nanoseconds, or roughly 600 instructions. Comparison with existing software is tricky, but this was at most one half the time it took with existing high performance user-mode libraries on that platform. One study [5] shows around 2000 cycles of overhead just to enter and leave the message I/O functions. We feel it is safe to say we are no worse than MPI for short message sends.

Sending packets quickly is not useful if the receiving end adds large latency. Since we are involving the operating system in I/O, the remote process is no longer spinlooping on a network interface register, but is rather blocked on a read system call. Even a process private system call may not be fast enough in this case.

What is required is operating system involvement in the implementation of the message operation, not just moving data to the process.

Active Message Support in the Interrupt Handler. Active Messages [6] are Remote Procedure Call (RPC)-like messages that are processed asynchronously to the destination process. In the original design,

Active Messages were tightly coupled to use of the SPMD model and contained the address of a function to run in the process. In some later systems, in which not all processes on all nodes have an identical address space, the function address changed to an integer defining the operation to run. In many applications the set of message types is limited to a small number of simple operations, such as fetch and add, compare and swap, and so on. The key different between Active Messages and RPC is that with an active message, the Active Message Handler receives the message and executes the function call rather than the application process.

To support low-latency OS involvement in message receiving, we implemented active messages in the interrupt handler. In order for this to work, the interrupt handler has to be able to access memory in the process. In standard systems, such access is not possible because an interrupt context operates outside the context of any process. In order to allow interrupt handlers to access process memory, the kernel must support a common address space between processes, the kernel, and the interrupt handler. We implemented such support via Single Address Space segments.

Single Address Space Segment. In the Single Address Space Segment (SAS) pointers are unique across all processes, the kernel, and the interrupt handler. In other words, in this address space, a pointer referenced in a process, kernel, or interrupt handler maps to the same physical address. Therefore, when two processes share memory in the SAS, they use the same pointer values. Sharing pointers is not related to any particular programming model, so that different processes running different code can share memory through the SAS. Access control rules apply in SAS just as they do for any other part of the process address space, allowing processes to control who sees data in the shared segment. *The availability of the SAS does not imply any diminution of standard OS memory protection mechanisms.*

Since an address in the SAS has the same meaning in all modes, including interrupt mode, higher level operations can now be processed in a very low level part of the network stack. We can implement very low latency message handling.

We combined the process private system calls, SAS, and Active Message Support to provide processes on Blue Gene with a high performance synchronization primitive. A basic ping-pong using this mechanism took about 2.5 microseconds on Blue Gene, or about 1/2 the time of MPI. We also implemented 'active rings', in which the packets are dequeued directly into a ring buffer in user space. Active rings are equivalent in function to the Infiniband queue pairs, but do not require the significant offload processing that Infiniband requires.

Transparent large page size. Translation Lookaside Buffer (TLB) overheads can have a significant impact on performance [7]. The TLB hides the cost of virtual to physical address translation by caching the most recently used page table entries. With small (4KiB) page size, TLB misses are more likely, requiring a costly walk of a multi-level page table. To reduce the number of pages and thereby increase the likelihood of a TLB hit, modern CPUs support large page TLBs, making it possible for the OS to provide multiple page sizes. We implemented large page support to reduce TLB overhead. On the Blue Gene, all heap pages were mapped to 1 MiB pages rather than the more standard 4KiB pages. On x86, our minimum page size is 2 MiB; in heaps larger than 1 GiB all pages above the 1 GiB boundary are 1 GiB. On x86, using 2 MiB pages makes the page table walk 33% faster and reduces the number of page table pages by a factor of 512.

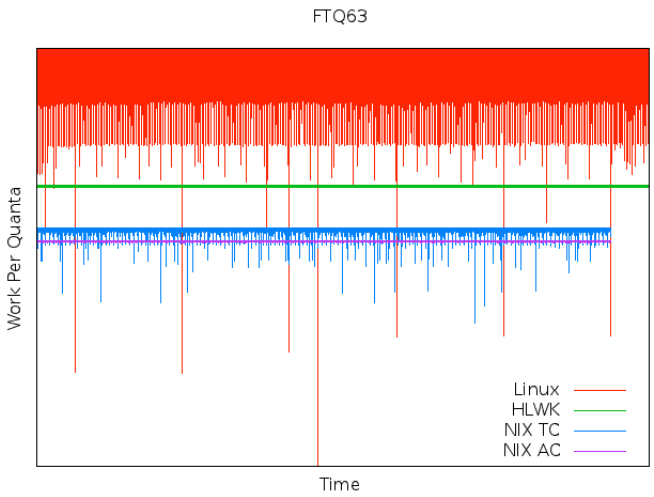

Figure 3: Comparative performance of FTQ on Linux, NIX timesharing core, and NIX application core. The flatter the line, the lower the interference.

2.2 Core specialization

We implemented role-based execution in NIX for homogenous and heterogenous SMPs, an idea motivated by our talks with CPU vendors about future directions in many-core architectures. For power and space reasons, some future many-core CPUs might have a large set of cores which only run user-mode. Further, it is possible that cores will have disjoint memory spaces – not all cores reference the same memory. To support role-based execution, NIX differentiates between cores that run full OS services and those that run dedicated application processes. Cores are designated as time-sharing, which are general purpose cores that can run multiple processes as well as OS kernel code; application, that can run only application code; or kernel, that are dedicated to running the OS kernel. Cores can be partitioned at boot time, which is natural for architectures with heterogeneous hardware cores, or dynamically, to accommodate homogeneous cores that are partitioned for performance. Partitioning cores based on hardware capability is needed in the case that cores are not able to run general purpose code or an OS. Dedicating a set of cores from a homogeneous pool to an application eliminates OS interference, which has been a continuing issue in HPC systems. When application code executes a system call, control is transferred to a kernel core.

On HPC node measurements, NIX showed very good performance, as discussed in [3]. We show results for FTQ [8] in Figure 3. FTQ is part of the Sequoia benchmark set [9] and measures the variation in amount of work achieved over time; the lower the variation, the better. In this case, the Linux performance, shown in red, is the worst. The NIX timesharing performance (green) is better, but hardly ideal. The application core (flat purple line) achieves theoretically perfect performance; there is no measurable OS noise, which makes sense as no OS code, and no interrupts, are active on application cores. Similar experiments performed with FusedOS showed the same behavior, validating our premise that core specialization with homogeneous cores maximizes compute-intensive application execution by eliminating OS noise.

Linux system call support. In the HPC community, Linux compatibility is no longer optional. As an extreme example, even the Blue Gene CNK, a lightweight kernel, provides extensive Linux system call support. It need not be complete: on the CNK, not all system calls are supported, and of those that are

supported not all possible permutations will work. Nevertheless, the set of system calls that has to be supported has grown over time.

While it is also possible to provide compatibility at the library level, experience shows that it is much easier to support Linux apps at the system call interface than to write compatible libraries.

NIX supports a limited number of Linux system calls on both Blue Gene and x86 systems. Linux processes are run by a manager process which reads the ELF binary into its address space, sets up context, and jumps to the entry point. The manager manages the process in the same way that the lguest hypervisor manages its guest OS [10].

Supporting the entire Linux system call set in the NIX kernel would be a massive undertaking. NIX provides support in one of two ways. For system calls requiring high performance, we support them directly in the kernel. System calls which are infrequent or very complex are 'bounced' out to the manager process via a signal. Because the process is mapped into the manager's address space, it can examine the process to see what it is doing; hence, passing pointers in system calls is not a problem.

2.3 FusedOS: core specialization with Linux and Blue Gene

The ultimate way to provide Linux support, of course, is to provide Linux. As part of an IBM Research team, we also participated in development of FusedOS [11], which follows the same basic principles as NIX, but uses Linux and the BG/Q compute node kernel as the foundation instead of Plan 9. The FusedOS prototype leverages Linux with small modifications on the Kernel Cores and implements a user-level light weight kernel called Compute Library (CL) by leveraging CNK on the Application Cores.

Table 1: LAAMPS Results

Environment	**1 Thread**	**16 Threads**	**64 Threads**
Linux	361.968	364.457	773.900
Standalone CNK	357.278	361.740	566.436
FusedOS Application Core	357.490	362.059	544.566

Table 1 shows the performance results from running LAMMPS benchmarks in the three operating environments. The results are the run-time for a single thread in seconds on the Bluegene/Q. As the table shows, all three examples have similar performance with a single thread (with CNK and Application Cores having a slight advantage over Linux), but as the number of threads grows, both native CNK and Application Cores demonstrate a significant performance advantage over Linux.

FusedOS is now available as open source on github.

Discussion: NIX vs. FusedOS approaches. Both NIX and FusedOS support the concept of application cores (ACs). Probably the most significant difference is in the way that ACs are managed. In NIX, while processes own an AC, the NIX kernel still manages the resources that the process uses, including memory and I/O; and, when a process performs a system call, it is a NIX or Linux system call. A NIX process is always a NIX process, whether running on a timesharing core or application core.

FusedOS implements a so-called cohabitation model, in which two kernels exist on the same hardware. FusedOS removes resources (including memory) from the domain of Linux completely: Linux can not even see most of the other kernel's memory. FusedOS sets up cores and starts applications running the CL on them. Those applications use Linux system call support only indirectly, via a control process which maps

application system calls to Linux requests, interpreting or mapping them if need be. FusedOS provides more flexibility in the application environment at the cost of having to write an OS environment for non-native applications.

2.4 Related OS research

One of the earliest examples of core roles was in the implementation of Sandia's Puma operating system [12]. The authors point out that "Usability is provided by creating node partitions specialized for user access, networking, and I/O." More recent work in this vein can be found in Kitten [13], which allows Linux binaries to run in a VM partition. However, the overhead in this environment is inverted from our systems; Linux binaries need a Linux kernel and VM under which to run, increasing the overhead and possible interference; whereas on our application cores, the overhead is greatly reduced.

Our early thinking was also informed by the Barrelfish work [14], which each socket runs a separate kernel. This work swims against the tide of ever-increasing core counts running under a separate image. Barrelfish does not differentiate core roles, however, and all sockets have at least one core dedicated to the OS.

Tesselation [15] advocates spatial partitioning as well, although each core does run a kernel.

Cray has recently added support for core specialization [16], and reports an improvement in performance where it is used. Their approach is very different from ours, however; they have observed that not all applications scale to use all cores, and they allow the application to assign MPI progress threads, running under the Linux kernel, to unused cores. Cray Core Specialization is the obverse of NIX and Fused-OS.

Lange [17] makes a case for a partitioned model much like FusedOS, though it is not clear how much has been implemented.

3 Task management with a reliable, distributed metadata store

The hybrid runtime (Figure 2) provides basic OS services such as protection, memory management, and safe access to low level communication hardware and external resources. To support fine grained, adaptive, dynamic, and massively parallel computational tasks, we designed and prototyped task management libraries that manage work distribution and the flow of data in extreme-scale systems. In prior work ([18–20]), we have demonstrated a data-driven computational model: a computation is organized as a collection of tasks with each task described by the data it requires from a global address space and a sequential function to evaluate the tasks. The scheduling of the tasks together with the communication is automatically handled by the runtime system. The data-driven model (Figure 1) combined with the global address space make it possible for task schedulers to migrate tasks during execution, which is required for both reliability and load balancing. There are many different ways to approach both global address space and task management. We now report on experiments with a variety of libraries. All these libraries are user mode and run as part of the application, where the application can run under an OS or, in some cases, as "bare metal." Several of the libraries additionally support "fault oblivious" execution, in which the application runs normally despite node failure or slowdown, with failure management handled purely within the library.

3.1 Task queues in a reliable distributed hash table with libFOX

Our first experiment built the libFOX API that uses an efficient, scalable, distributed hash table to store global task metadata. We adapted an existing distributed key-value store from the commercial sector to the HPC environment. Memcached [21] from the Couchbase project [22] is one candidate of many NoSQL systems [23] in active development based on key-value pairs. Memcached has demonstrated scalability to tens of thousands of nodes on enterprise servers. A useful feature of the Couchbase version of Memcached is a

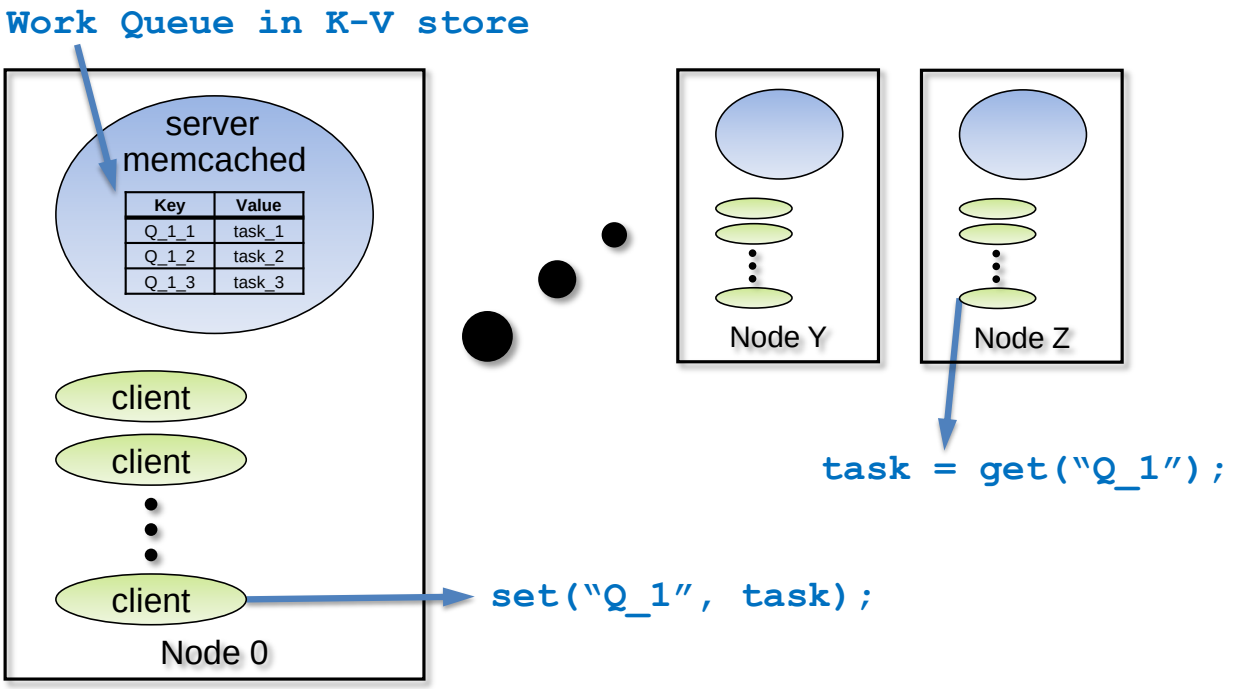

Figure 4: Task queue implemented with key-value pairs in memcached.

pluggable storage engine that is separated from the main TCP/IP server. This feature facilitates experimentation with various storage and caching implementations without affecting the main server. Additionally, Memcached includes reliability services such as replication and relocation of data store servers. In addition to fast prototyping of a task management API, we wanted to know whether a widely used, highly efficient enterprise level distributed key/value store was suitable as infrastructure on which to base HPC services.

For the HPC environment, both task queues and data can be represented in a key-value store (see Figure 4). Even though keys and their associated values are fundamentally independent, they can be organized hierarchically by including references to other key-value pairs. For example, a task queue is constructed with front and back pointers stored in key-value pairs that reference work units stored in other key-value pairs.

Counting semaphores for fast synchronization. A fundamental feature that is absent from Memcached is the ability to synchronize efficiently between producer and consumer processes. Using the standard Memcached interface, a worker needs to poll for a task to appear in a task queue through repeated get operations on a specific key. A solution to this problem is to introduce distributed counting semaphores. With counting semaphores, processes that write data on a key can signal processes waiting for data on a key. Distributed counting semaphores are implemented with the addition of a count field to key-values pairs. Synchronous set and get operations pass a delta parameter that increments or decrements the semaphore count. A synchronous get operation waits until the count is at least equal to the requested delta and then returns the data associated with the key. Synchronizing through a key allows a replicated server to support the semaphore operation in the advent of failure.

We developed libFOX to provide Memcached clients with a task queue abstraction and other parallel communication patterns common to HPC. These patterns include the broadcast of parameters to workers, work distribution, and the collection of results. Enhancing the Memcached interface to support distributed

counting semaphores made libFOX more efficient for task distribution and management by reducing the need to poll.

MCphoton. A simple Monte Carlo radiative heat transfer simulation called mcphoton was chosen to study how an HPC application might interact with a distributed data store through a task model with the potential for resilience. A highly parallel version was developed that distributes tasks through the task queues provided by the libFOX interface. An implementation that uses Memcached for all communication has shown near linear performance scaling on a cluster up to 256 nodes and 2KiB cores. The current implementation of Memcached uses sockets for communication and encounters scaling problems beyond 2KiB cores, highlighting the differences in latency requirements for enterprise vs. HPC. To improve efficiency, we developed a distributed key-value store EbbRT using native HPC communication primitives on the Blue Gene/Q.

3.2 Elastic Building Blocks (Ebb)

The libFOX library targets a global task queue using a key/value store based on a distributed hash table to store task metadata. To more generally support future HPC applications that need a combination of customized distributed runtimes and general purpose commodity operating systems, we have constructed a prototype runtime, the Elastic Building Block Runtime (EbbRT). EbbRT provides the Elastic Build Block (Ebb) object model which enables developers to encapsulate distributed components of software and exploit distributed data structures and associated communication optimizations in the face of dynamic changes to the set of nodes. We used the EbbRT infrastructure to implement the libFOX API. Additionally we prototyped a fault tolerant, data flow driven implementation of the mcphoton application with EbbRT.

Given a system wide identifier for an instance of an Ebb called an EbbId, a client can invoke an Ebb through a well defined interface. For example, a hash table Ebb may provide `get(key)` and `put(key, value)` functions. Hidden from the client and based on programmer defined behavior, an Ebb internally constructs *representatives* of itself on nodes and cores on which it is accessed. The Ebb programmer must then have representatives communicate with each other as necessary to satisfy client requests. This model enables Ebb programmers to design highly tuned implementations of common interfaces and allows application developers to select appropriate implementations.

EbbRT is a thin portable library which provides the basic interfaces and language bindings to allow developers to instantiate and manage Ebb instances. We have developed an implementation that can be linked with Linux applications as well as an implementation that can run bare metal on x86-64 and PowerPC64 machines (including Blue Gene). EbbRT is open-source, written in C++11, and is under active development at https://github.com/SESA/EbbRT.

All instances of EbbRT form a distributed structure that allows Ebb instances to cross the boundaries between the bare metal nodes and the Linux nodes. To concretely understand this, consider an Ebb that implements a file system interface (eg. `open`, `fstat`, `mkdir`, etc.) that can be used directly by application code or behind a libc wrapping. When invoked by an application process on a Linux node the Ebb will use a local representation of itself that utilizes the underlying Linux interfaces. On bare metal nodes, however, the Ebb employs a specialized bare metal representative that serializes the operation and data as necessary and sends them to a Linux representative. The distributed bare metal representative and Linux representative structure and relationships are hidden from the callers. Furthermore, the representatives can optimize various interactions for the sake of improving performance and or fault-tolerance. This can be done

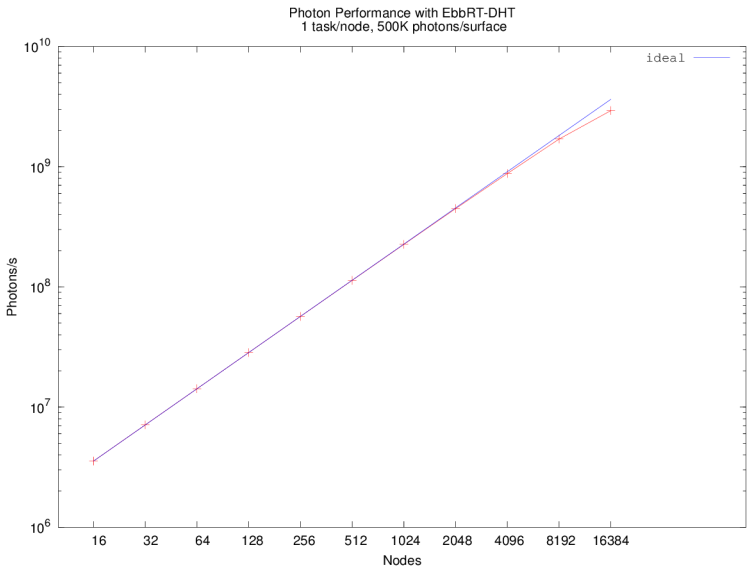

Figure 5: Performance of Ebb-based libFOX compatible version of mcphoton on BG/Q 1-16 Racks.

via any appropriate methods as needed (eg. partitioning and replicating data, exploiting hardware features, optimizing read vs write access ratios, etc.).

In addition to the object model EbbRT provides a non-blocking event-driven execution model. External events due to machine failures, network traffic, timers, etc. execute programmer-specified event handlers to completion before further events are processed. In the context of bare metal execution, these events execute at supervisor level and therefore allow for software to react efficiently and with low latency to asynchronous behavior.

A critical aspect of the EbbRT approach is to allow for the easy elimination of overheads by exploiting the dedicated nature of HPC applications while tailoring the solution to existing application code. Our work has focused on exploring how EbbRT could be leveraged in implementing the mcphoton simulation code. We explored two approaches: 1) porting the FOX memcached data store version to use an Ebb-based implementation of the data store with minimal application changes and 2) the construction of an Ebb runtime that permits a re-implementation of mcphoton to exploit both a task model and a data-store that achieves application-level fault oblivious execution. Both these approaches are discussed below.

mcphoton in EbbRT. The libFOX library was developed to act as a generic HPC centric application interface to a distributed data store. As such, it provides a natural separation between application code and the software implementing the underlying data store. LibFOX based applications provide an ideal setting to explore and contrast the EbbRT model for HPC programs compared to how one might naturally develop a libFOX program using a commodity distributed data-store and associated OS infrastructure.

Our first approach focused on preserving the application code and injecting calls to Ebbs behind the libFOX API. The existing mcphoton application used libFOX implemented on top of memcached. We re-implemented libFOX on top of an Ebb distributed hash table developed for and tuned to use the network of a supercomputer. This allowed the mcphoton application to run on the Bluegene/Q Mira without any application modifications.

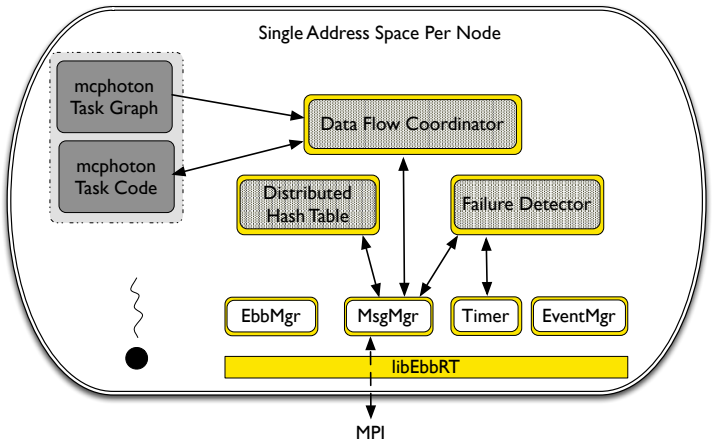

Figure 6: Elastic Building Block Runtime for libFOX based version of mcphoton.

We evaluated this version on runs from 1-16 racks of Mira, The Argonne Blue Gene/Q (see Figure 5). Ideal scaling was observed to 8 racks (8192 nodes), at which point the overheads of the less-scalable portions of the libFOX interface (e.g., implementation of collective operations) began to dominate the performance. Our goal for this task was to port an application with minimal application change, and demonstrate increased levels of performance and scale. From this perspective we found that the Ebb approach (with appropriate library and component design) does permit encapsulation of distributed system level components that can be directly linked into the application. Scalability greatly improved over the initial fast prototype build using memcached, but we eventually encountered scalability limitations due to the libFOX implementation of collective operations that used naive point-to-point communication.

Fault-tolerant mcphoton. We next built a fault oblivious mcphoton application in which the application continued to operate normally in the presence of faults, with the underlying Ebbs providing fault management. To this end, we developed a custom EbbRT task runtime and associated data-store that encapsulates fault-tolerant behavior, and yet allows us to easily implement the mcphoton application. We implemented a new version of mcphoton using an Ebb runtime constructed specifically for task-driven programming that allowed us to explore encapsulation of fault-tolerance and improved scalability properties as illustrated in figure 6.

We modified the mcphoton application to generate a data flow graph representing the computation to a DataflowCoordinator Ebb. The graph expresses the dependencies between fine grained tasks (which outputs are inputs to which tasks). Figure 7 shows the performance characteristics of this version. The Dataflow-Coordinator Ebb schedules individual tasks whose inputs have been produced. The individual tasks can run on any node in the system and the generated outputs are stored locally. If a node is detected to have failed, then the DataflowCoordinator can reschedule any tasks whose data needs to be reproduced and the computation can proceed without the failed node. We implemented a centralized DataflowCoordinator that stores all task and data location information (but not the actual data) on one node and uses a distributed hash table Ebb to store the actual data. We detect failures using an unreliable failure detector (which can give false positives) called the Phi Accrual Failure Detector [24]. To stress our implementation we configured mcphoton with very fine grain tasks (on the order of one second each). Our implemented runtime was

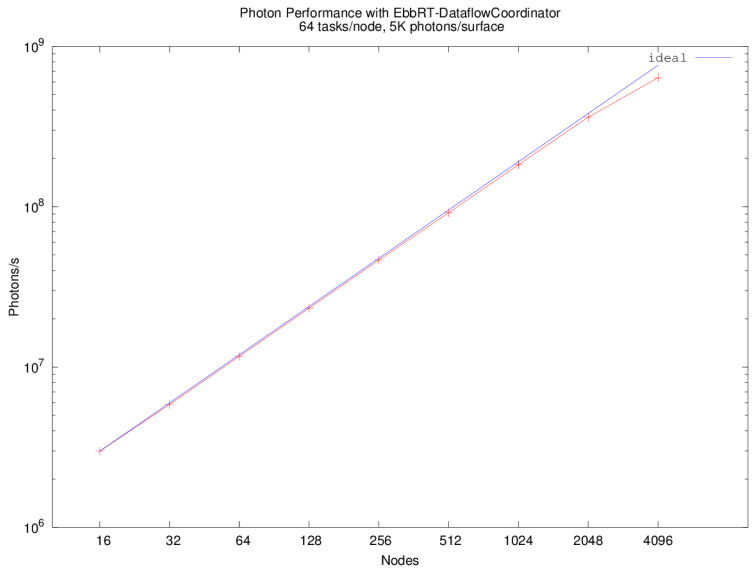

Figure 7: Performance results of fault tolerant dataflow implementation of mcphoton on BG/Q 1-4 Racks.

able to run the mcphoton computation with perfect weak scaling up to 4096 nodes with no failures. The computation tolerates multiple failures to nodes which do not store the task and data location information (i.e., the dedicated DataflowCoordinator node.)

Our goal was to demonstrate the ability to build higher level run-times using EbbRT by building reusable system software components. Our construction of the libFOX key/value store, task library, dataflow coordinator, and failure detectors, all within the Ebb framework, gives us confidence that this is a fruitful approach.

For future work, there are a number of potential improvements to our data dependent runtime to improve both fault tolerance and scalability. One could design a DataflowCoordinator that replicates the task and data location information to multiple nodes in order to provide additional fault tolerance. Additionally the information could be partitioned (having portions of the data flow graph managed by separate nodes or groups of nodes) to provide better scalability.

3.3 Tuple space task library

Another aspect of our research into fault oblivious task management shifted from imperative to declarative programming interface. Traditional imperative frameworks give more control to the application programmer, potentially allowing tighter optimizations to be applied. As fault-tolerance becomes a greater concern for HPC, declarative styles become more appealing. Using a declarative interface, the programmer merely prescribes the work to be done, and responsibility for scheduling and doing the work is handled by a runtime library.

Our work is inspired by the tuple-based frameworks Linda [25] and Concurrent Collections (CnC) [26], examples of "coordination languages." Rather than using explicit message passing, processes coordinate through a globally visible key-value store. Workers are decoupled in space and time. MPI may seem more intuitive for certain tightly-coupled physics problems. However, the model of decoupled workers not only provides obvious advantages for resilience, it gives the runtime a chance to optimize performance in ways that

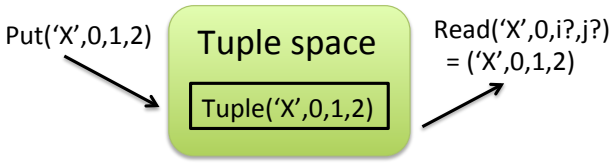

Figure 8: Basic tuple operations showing use of wildcard operators.

might be very difficult to code by hand for each application. In tuple-space systems a small set of primitive operation (read, pull, put) are available, simplifying the underlying runtime. For example, in Linda, process 0 "puts" data, which is eventually "pulled" by process 1 (Figure 8). The tuples are arbitrarily typed. A powerful feature in Linda are wildcard reads, shown by a question mark, allowing any matching tuple to be returned. Linda thus intrinsically supports data-flow execution, operating on data that are *ready*.

FOX-tuple Building on the underlying data-flow ideas of Linda, we have constructed a task-based coordination framework. Linda has been the focus of significant fault-tolerance research, with numerous methods already established. Rather than design a system from scratch, we sought to adapt Linda-like ideas into an asynchronous task framework, thereby giving us a wealth of fault-tolerance strategies on which to build. Tasks are created, with inter-task dependencies being expressed as tuples in a specific tuple space. In contrast to Linda, the FOX-tuple framework has a more restricted, well-defined style. However, it provides a rich set of semantics and library functions. Tasks and dependencies are declared via simple API calls, here shown for matrix multiplication.

```
fox_dependency_declare(matrix_block,
    char, int, int, int //label, iteration, row, column
    ArrayData); //FOX array type
fox_tuple_task_declare(multiply,
    int, matrix_block, matrix_block, matrix_block) ;
fox_append_migrate_dependency(multiple,matrix_block);
...
```

Dependency tuples must be declared and the arguments for the tasks declared. At runtime, dependencies are appended to the tasks with a specific type. At the core, only the three primitive operations are used. However, they are wrapped inside higher level API calls, allowing the user to simply declare dependency types - read, migrate (systolic arrays), in-out (reduce). The runtime automatically performs the required sequence of primitive operations.

The key performance obstacles are related to communication. In MPI, data is exchanged directly between processes. Using a distributed key/value store, processes must indirectly exchange data via the store, increasing communication cost. The FOX-tuple framework is implemented as a distributed hash table, with tuples being mapped to a node carrying the corresponding subset. Large blocks of data are never transferred off-node. An RDMA metadata handle is placed into the key-value store instead. To retrieve large data blocks, the metadata is first read from the distributed hash table, and the data block is filled by an RDMA get directly from its location. All these details are hidden from the programmer.

While the indirection might seem to increase communication costs relative to MPI, the extra cost is small. For large data blocks, MPI uses a rendezvous protocol. A sender must first send a metadata header from which the receiver performs an RDMA get. The send/recv pair is finished with an MPI ack. MPI, despite

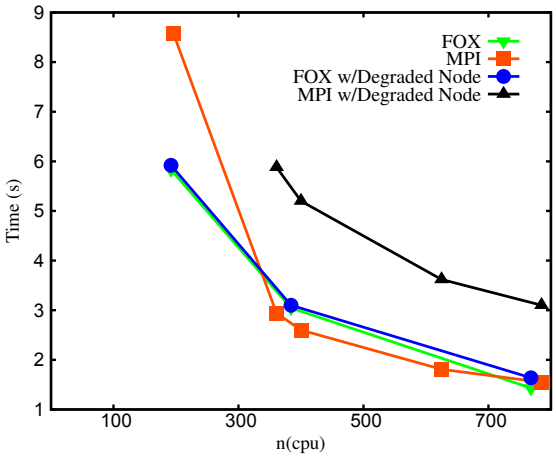

Figure 9: Performance of FOX-tuple framework and MPI for matrix-matrix multiplication.

being direct communication, therefore requires multiple message exchanges. FOX therefore directly transfers large blocks with only minimal extra latency to negotiate the transfer.

For the case of matrix-matrix multiplication (Figure 9), we compared the performance of FOX to MPI on the Cray XE6 Hopper [27]. Despite being "declared" rather than explicitly written step-by-step, the FOX framework is very competitive with MPI. When certain nodes are degraded, running at half speed, MPI has no ability to rebalance around the faults. In contrast, the FOX-tuple performance remains stable.

4 TASCEL work-stealing task library

Our final task library considered the design of a more traditional task-parallel runtime system, pushed to extreme scale. Specification of parallel tasks exposes the computational cost and data-access relationships to the runtime and system software. We considered runtime approaches to automatically load balance a task-parallel computation, track the actions of the scheduler, and recover from faults in a scalable fashion.

In order to expedite our investigation, these were built on top of an active message library implemented using MPI. MPI allowed us to evaluate these ideas on today's systems at full-machine scales without requiring permissions to deploy a new software stack. The active message library allowed us to decouple our algorithms from the specifics of MPI. The task-parallel runtime system can be ported to the new software components described thus far in this paper by porting the active message library.

The application begins execution as a multi-threaded MPI program and enters a task-parallel phase (e.g. the loop body of an iteration) with one or more tasks. We support processing a fixed list of tasks or allowing tasks to spawn additional tasks. The task-parallel phase terminates when all initial tasks and any tasks transitively spawned by them have been processed. Upon termination of a task-parallel phase, the computation returns to the multi-threaded MPI mode. In this work, we used the MPI mode to set up the task-parallel phase, and focused on scalable fault-tolerant task execution within each phase.

4.1 Dynamically Load Balancing Iterative Computations

Applications often involve repeated execution of calculations with identical or slowly evolving execution characteristics. Such iterative applications often exhibit sufficient variation across iterations to preclude efficient static load balancing. This necessitates the use of dynamic load balancing approaches that incre-

mentally rebalance the computation over successive iterations.

We studied the design of load balancers for task-based iterative programs. We exploited the fact that the execution characteristics of iterative applications evolve over time, with significant *persistence* of such characteristics between successive iterations. Persistence-based load balancers measure task execution profiles in a given iteration and use that to improve load balance in the next iteration. We designed a hierarchical persistence-based load balancing algorithm that attempts to localize the rebalance operations and migration of tasks. The greedy algorithm rebalances "excess" load offered by a processor rather than attempt an optimal partition. This allows efficient implementation of the load balancing algorithm at the cost of potentially increased load imbalance. In addition, the greedy approach can potentially better retain data locality and topology-awareness from previous iterations.

Work stealing is an alternative dynamic load balancing approach to fix load imbalance within an iteration or a phase. In particular, under work stealing, a processor without work attempts to steal excess work from other processors until all execution terminates. This approach is especially beneficial when a phase can incur significant load imbalance and cannot be fixed through static or profile-guided schemes. Work stealing algorithms employ random stealing to efficiently redistribute work. While shown to be effective in theory and practice, such stealing interferes with data locality and topology-aware optimizations by dispersing work as the iterations progress. We developed a work stealing algorithm for distributed memory systems based on active messages. The algorithm acknowledges the costs incurred on distributed memory systems by minimizing round trip latencies, and the duration of locked operations. We designed *retentive work stealing* so as to reuse the task mappings resulting from work stealing in iterations. This implicitly retains data locality optimizations from prior iterations.

We demonstrated consistently high efficiencies on ALCF Intrepid, NERSC Hopper, and OLCF Titan for the Self Consistent Field (see Figure 10) and Tensor Contraction benchmarks on over 100K processor cores [28]. We observed that the hierarchical persistence-based load balancer achieves load balance comparable to the optimal centralized scheme in practice while incurring low overheads. We demonstrated scalability of work stealing at over an order of magnitude higher scale than prior published work. Retentive work stealing is also shown to further improve load balance while reducing stealing overheads, as the execution becomes increasingly balanced. Retentive stealing thus combines the benefits of persistence-based load balancing with work stealing's ability to quickly react to load imbalance.

Based on the demonstration of scalable load balancing, we ported a homology detection framework on TASCEL [29]. The implementation employed distributed memory work stealing to effectively parallelize optimal pairwise alignment computation tasks. This implementation was evaluated on up to 131,072 cores of the Intrepid IBM BlueGene/P system.

Related work Persistence-based load balancers have been extensively studied and employed in the context of Charm++ [30–32, 32, 33]. Unlike these efforts, we focus on the design of a greedy rebalancing algorithm. Work-stealing based load balancers have been extensively studied in Cilk [34, 35] and subsequent efforts [19, 36, 37]. Unlike these efforts, we reuse the scheduling information across iterations to demonstrate scaling at much larger core counts than in prior work.

4.2 Characterizing Work Stealing Schedulers

The flexibility inherent in work stealing when dealing with load imbalance results in seemingly irregular computation structures, complicating the study of its runtime behavior. We developed an approach to

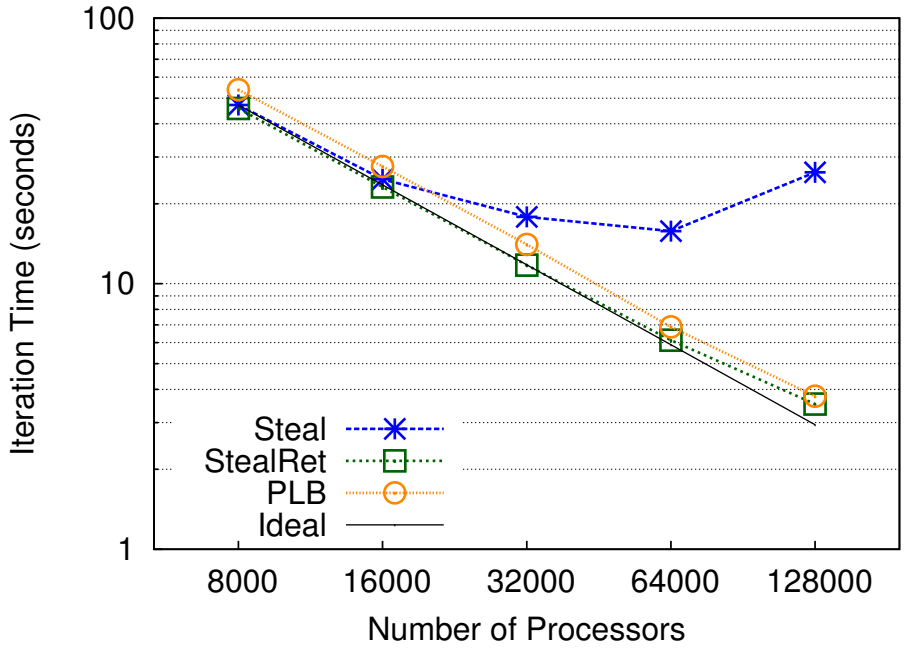

Figure 10: Execution time for the Self Consistent Field benchmark under traditional work stealing, retentive stealing, and persistence-based load balancing after convergence on OLCF Titan.

efficiently trace async-finish parallel programs scheduled using work stealing [38].

We consider two scheduling policies for async-finish [39] task-parallel programs. In the work-first policy, a processor, upon encountering a task to execute, pushes the currently executing task onto its local deque of tasks and begins to execute the new task. A thief can steal a partially-executed task pushed onto the deque. This policy mirrors the sequential execution order. In the help-first policy, the working thread continues to execute the current task, pushing any encountered concurrent tasks onto the deque. Once the current task's execution has finished, the processor extracts the task last enqueued onto its local deque to continue execution.

We identified key properties of both schedulers that allow us to trace the execution of tasks with low space and time overheads. These are used to construct a steal tree that tracks the steal operations efficiently. Implementations of these algorithms were evaluated on both shared and distributed memory systems. Figure 11 shows the trace sizes per core for help-first (HF) and work-first (WF) scheduling of SCF and TCE benchmarks on up to 32,000 cores. We observe that the trace sizes are small enough, amounting to less than 1KB per core in most cases, allowing effective storage and analysis of complete traces even at large core counts.

We demonstrated the broader applicability of this work, in addition to replay-based performance analysis, through two very different use cases: the optimization of correctness tools that detect data races in async-finish programs; the design of load balancing algorithms that exploit past load balance information to incremental adapt to changes.

Raman et al. [40] perform data race detection by building a dynamic program structure tree (DPST) at runtime that captures the relationships between the tasks. The DPST is built dynamically at runtime by inserting nodes into the tree in parallel. To detect races, any two computation steps that access the same memory location are checked whether they could execute in parallel in any possible schedule. If they do,

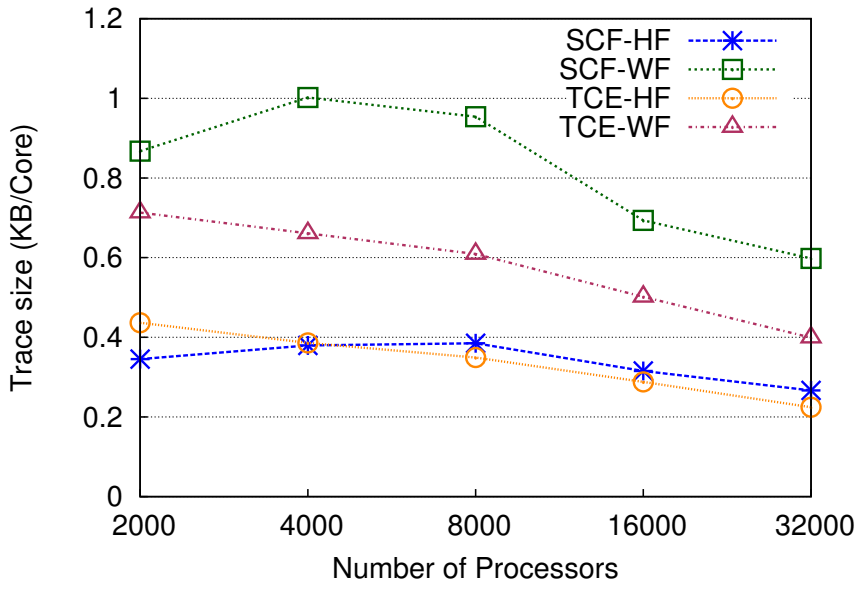

Figure 11: Size of traces for the SCF and TCE benchmarks scheduling using help-first (HF) and work-first (WF) scheduling policies on OLCF Titan.

a data-race is reported. A key step in the data race detection process is the determination of the lowest common ancestor to the two computation steps being considered. We observe that the steal tree that tracks the steal relationships can be used to speedup this computation. Experimental evaluation demonstrated that the cost of this operation was reducing by up to 80%.

We also demonstrate that the traces enable retentive stealing for recursive parallel programs. Our prior work on retentive stealing, discussed above, relied on explicit enumeration of tasks. This increases storage overheads and becomes infeasible when dealing with tasks with dependences. We exploit the fact that each node in the steal tree corresponds to a working phase and can be used as the starting schedule for subsequent execution in an iterative application. We thus extend the replay algorithms to allow further dynamic load balancing a steal tree. This was shown to further reduce the memory requirements of retentive work stealing.

Related work Series-parallel relationships in fork-join parallel applications have been exploited to optimize data-race detection [41, 42] and conflict detection in transactional memory systems [43]. However, unlike our scheme, these require global synchronization or locks to track the relationships and only support the work-first scheduling policy.

4.3 Selective Fault Recovery

Checkpoint-restart approaches to fault tolerance typically roll back all the processes to the previous checkpoint in the event of a failure, a heavyweight solution that will not scale to exascale. We developed novel data-driven resilience algorithms for work stealing schedulers that minimize both the overhead in the absence of faults and the performance penalty incurred by a fault [44]. We tracked the data operations to construct an idempotent data store. We simulated node failure by making all threads on a node non-

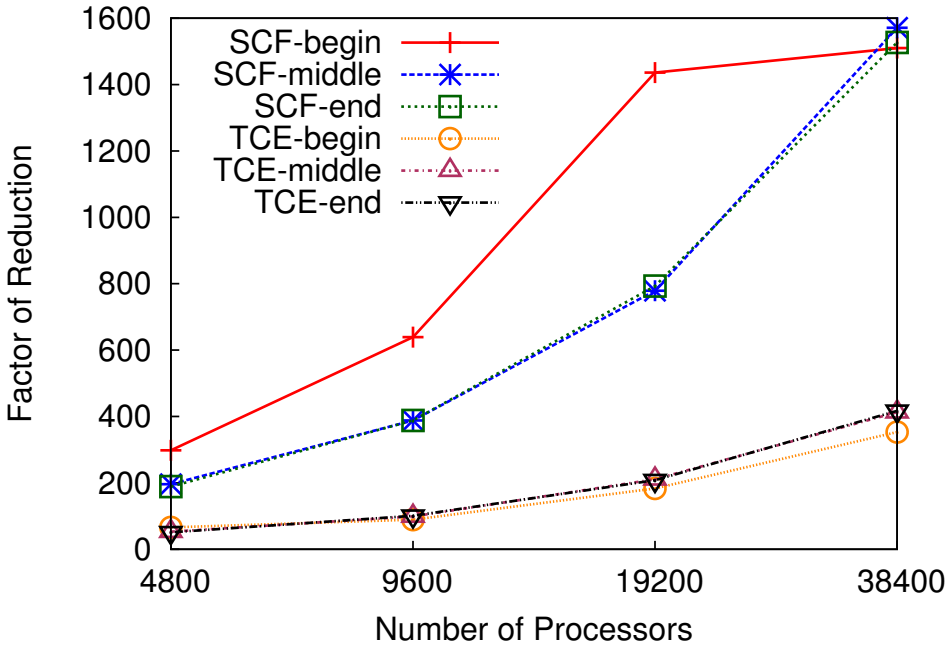

Figure 12: Factor of reduction in the number of re-executed tasks as compared to checkpoint-restart for SCF and TCE benchmarks when faults occur at the beginning, half-way through, and towards the end of execution.

responsive to all incoming active messages except termination of the task-parallel phase. This allowed us to evaluate the behavior of selective fault recovery without worrying about limitations of fault tolerance support in MPI implementations on today's parallel computing platforms.

We presented three recovery schemes that present distinct trade-offs: lazy recovery with potentially increased re-execution cost, immediate collective recovery with associated synchronization overheads, and non-collective recovery enabled by additional communication. We demonstrated that the overheads (space and time) of the fault tolerance mechanism are low, the costs incurred due to failures are small, and the overheads decrease with per-process work at scale. Figure 12 shows the factor of reduction in the number of tasks to be re-executed for the SCF and TCE benchmarks for various points in execution at which all threads in a compute node fail. We observe that selective recovery that exploits the characteristics of the resilient data store and task-parallel scheduler can significantly improve the penalty of a fault as compared to recovery based on collective rollback.

Related work Checkpoint and collective restart has been extensively studied [45–47]. While broadly applicable and easy to integrate into existing applications, these approaches typically incur the cost of coordination or message logging, checkpoint storage costs, and lost work from rollback. Alternatives such as shadow processes [48] overcome these costs while requiring redundant processes and communication.

Approaches to reduce these overheads employ techniques that target specific layers of the software stack. Fault tolerant MPI [49, 50] focuses on the development of a resilient communication library. Algorithm-based fault tolerance approaches target the development of fault tolerant parallel computational libraries that exploit the algorithmic properties of individual kernels, such as matrix-matrix multiplication and one-sided factorization [51,52]. Our approach to selective fault recovery exploits the properties of a computational

idiom.

Cilk-NOW [53] tolerates faults in recursive applications organized as *return transactions* with updates passed as function return values. The recovery mechanism re-executes entire execution sub-trees of a failed task. Our approach recovers individual tasks while supporting direct updates to data in global address space.

5 Conclusions

Our work investigates the system software stack for efficient exascale execution. On the OS front, we implemented new capabilities in novel research Operating Systems. Anticipating the need for OS involvement in the exascale environment we developed optimizations to make a general purpose OS competitive with library or lightweight kernel performance. NIX aimed at getting the kernel more, rather than less, involved in the computation's I/O activities. NIX explicitly excluded OS bypass as a means of moving network data and achieved performance competitive with OS bypass for critical operations. We believe that to achieve power and thermal management goals, the kernel must be included, not bypassed.

Core specialization allows an operating system to assign roles to cores, the most common use being partitioning cores to application only or timeshared. We presented two implementations of core specialization. In each case, core specialization improved application performance. In NIX, cores could have timesharing, application, or kernel roles. An application core ran as a NIX process. In contrast, FusedOS ran Linux on a couple of cores to manage at the CPU socket level and supported a traditional HPC OS on the other cores, allowing CNK programs to run unchanged under FusedOS.

Both systems supported Linux system calls because, in today's current HPC environment, some level of support of the Linux system call ABI is required. NIX supported a unique model in which some level of Linux system call support is done in the kernel, and some in a user-mode handler. FusedOS continued the Blue Gene tradition of function shipping system calls to a Linux kernel. Of the two, FusedOS provides more complete support, at cost of having to run a full Linux kernel on each node. The ideas prototyped by these research OSs are being incorporated into next generation OS projects such as UC Berkeley Akaros [54] and University of Tokyo IHK [55].

In the runtime area, we studied the use of four distinct frameworks. Building on the substrate of a reliable data store, we prototyped and evaluated a task management library using enterprise key-value store; an object-oriented framework for distributed HPC task and fault management services; a declarative tuple space abstraction adapted to the HPC regime, and a highly scalable and fault tolerant task library used by parallel phases of traditional iterative computation. Each library provides a different programming abstraction, yet they all follow common themes of reliable data store, dynamic load balancing, and fault oblivious execution. While there are now a multitude of task management libraries available in the HPC community, our approach integrated fault oblivious execution into the library from the outset. We quantitatively evaluated alternative fault management strategies and implementations, and identified performance overheads and scaling limits. The TASCEL library scaled to hundreds of thousands of cores and to two very different supercomputing architectures.

Much of our software is available as open source, and concepts explored and evaluated in FOX are being pursued in on-going advanced OS/Runtime projects such as Argo [56] and Akaros [54].

References

[1] "The opportunities and challenges of exascale computing." Retrieved 2013/01/10.

[2] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, pp. 221–254, Summer 1995.

[3] E. Van Hensbergen, R. Minnich, J. Mckie, and C. Forsyth, "Hare: Final report," *RC25241, IBM Corp*, 2011.

[4] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador, "Nix: A case for a manycore system for cloud computing.," *Bell Labs Technical Journal*, vol. 17, no. 2, pp. 41–54, 2012.

[5] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk, "The importance of non-data-communication overheads in mpi," *International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 5–15, 2010.

[6] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Active messages: a mechanism for integrated communication and computation*, vol. 20. ACM, 1992.

[7] C. McCurdy, A. L. Cox, and J. Vetter, "Investigating the tlb behavior of high-end scientific applications on commodity microprocessors," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pp. 95–104, IEEE, 2008.

[8] M. Sottile and R. Minnich, "Analysis of microbenchmarks for performance tuning of clusters," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 371–377, IEEE, 2004.

[9] "Asc sequoia benchmark codes." Retrieved 2013/12/30.

[10] R. Russell, "Lguest: implementing the little Linux hypervisor," in *Linux Symposium 2007*, June 2007.

[11] Y. Park, E. V. Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski, "FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, pp. 211–218, 2012.

[12] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. McCabe, and R. Riesen, "A system software architecture for high end computing," in *Supercomputing, ACM/IEEE 1997 Conference*, pp. 53–53, IEEE, 1997.

[13] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, *et al.*, "Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[14] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *Proceedings of the Workshop on Managed Many-Core Systems*, p. 27, 2008.

[15] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client os," *HotPar09, Berkeley, CA*, vol. 3, p. 2009, 2009.

[16] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the cray linux environment core specialization feature to realize mpi asynchronous progress on cray xe systems," *Proceedings of Cray User Group*, 2012.

[17] J. Lange, "Partitioned multistack evironments for exascale systems,"

[18] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, (New York, NY, USA), ACM, 2008.

[19] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, (Portland, Oregon), November 2009.

[20] G. Cong, S. B. Kodali, S. Krishnamoorthy, D. Lea, V. A. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *ICPP*, pp. 536–545, 2008.

[21] "Memcached." Retrieved 2013/12/13.

[22] "Couchbase server." Retrieved 2013/12/13.

[23] "Not only sql (nosql) databases." Retrieved 2013/12/13.

[24] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pp. 66–78, IEEE, 2004.

[25] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman, "The Linda Alternative to Message-Passing Systems," *Parallel Comput.*, vol. 20, pp. 633–655, 1994.

[26] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "The Concurrent Collections Programming Model," Tech. Rep. TR 10 12, Department of Computer Science, Rice University, 2010.

[27] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, "A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI," in *PMBS '11: 2nd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, 2011.

[28] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 137–148, ACM, 2012.

[29] J. Daily, S. Krishnamoorthy, and A. Kalyanaraman, "Towards scalable optimal sequence homology detection," in *Workshop on Parallel Algorithms and Software for Analysis of Massive Graphs (ParGraph)*, pp. 1–8, IEEE, 2012.

[30] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA'93*, pp. 91–108, September 1993.

[31] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[32] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *IJHPCA*, 2010.

[33] H. Menon and L. V. Kalé, "A distributed dynamic load balancer for iterative applications," in *SC*, p. 15, 2013.

[34] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *PPoPP*, pp. 207–216, July 1995.

[35] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *USENIX*, pp. 10–10, 1997.

[36] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High performance remote memory access communication: The ARMCI approach," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 233–253, 2006.

[37] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *PPoPP*, pp. 201–212, 2011.

[38] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé, "Steal tree: low-overhead tracing of work stealing schedulers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 507–518, 2013.

[39] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 519–538, ACM, 2005.

[40] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 531–542, 2012.

[41] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, "On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pp. 133–144, 2004.

[42] T. Karunaratna, *Nondeterminator-3: a provably good data-race detector that runs in parallel.* PhD thesis, Massachusetts Institute of Technology, 2005.

[43] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pp. 163–174, 2008.

[44] W. Ma and S. Krishnamoorthy, "Data-driven fault tolerance for work stealing computations," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 79–90, ACM, 2012.

[45] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[46] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX Annual Technical Conference*, 2007.

[47] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," in *Journal of Physics: Conf. Series (SciDAC)*, vol. 46, pp. 494–499, June 2006.

[48] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 44:1–44:12, 2011.

[49] G. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 1908, 2000.

[50] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pp. 1–18, 2002.

[51] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. C-33, pp. 518 –528, June 1984.

[52] S.-J. Wang and N. Jha, "Algorithm-based fault tolerance for FFT networks," *Computers, IEEE Transactions on*, vol. 43, pp. 849 –854, Jul 1994.

[53] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 1997.

[54] "Akaros." Retrieved 2014/01/20.

[55] "Interface for heterogeneous kernel." Retrieved 2014/01/20.

[56] "Argo exascale os/r." Retrieved 2014/01/20.