# FLUX: A Next-Generation Resource Management Framework for Large HPC Centers

Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, Martin Schulz

Lawrence Livermore National Laboratory,
Computation Directorate,
Livermore, CA 94550
{ahn1,garlick,grondona1,lipari1,springme,schulzm}@llnl.gov

***Abstract*—Resource and job management software is crucial to High Performance Computing (HPC) for efficient application execution. However, current systems and approaches can no longer keep up with the challenges large HPC centers are facing due to ever-increasing system scales, resource and workload diversity, interplays between various resources (e.g., between compute clusters and a global file system), and complexity of resource constraints such as strict power budgeting. To address this gap, we propose Flux, an extensible job and resource management framework specifically designed to deal with the requirements of next-generation HPC centers. Flux targets an entire computing facility as one common pool of diverse sets of resources enabling the facility to accommodate site-wide constraints (e.g., for power limits). Yet, its scalable and distributed design still offers scalable and effective scheduling strategies. This paper details the design of Flux and describes and evaluates our initial prototyping effort of the key run-time components. Our results show that our run-time prototype provides strong and predictable scalability.**

***Keywords*—resource management, communication framework, run-time, key value store, scalable process management services.**

## I. Introduction

Scalable Resource and Job Management Software (RJMS) is critical for High Performance Computing (HPC). It is the centerpiece that enables efficient execution of HPC applications while providing the compute center with the means to maximize the utilization of its diverse computing resources [1]. However, current state-of-the-art RJMS systems are becoming increasingly ineffective in dealing with the growing diversity and size of resources fielded in HPC centers today. Large-scale resources are no longer limited to a few extremely large individual machines like Sequoia (LLNL) [2], but are becoming commonplace through all computing resources sited at a center, including commodity Linux clusters. A modern RJMS must provide management services to all of the resources at the center, while maintaining scalability, low noise, fault tolerance, and enforcing global constraints.

In addition, greater difficulties in code development on larger systems impose increasingly challenging requirements on debugging, tuning, testing, and verification as well as new techniques for accurate correlations between user-level errors and system-level events. These tools require adequate RJMS support [3], [4], [5], [6], [7] for launching of daemons, allocation of analysis resources, or the ability for secure third-party access to running jobs. Unless the RJMS can effectively address these requirements, HPC users may suffer significant productivity losses across the full application development life-cycle.

The traditional resource and job management paradigm for large HPC centers organizes the resources sited at a center in a static and flat hierarchy. Typically, it runs an RJMS instance such as SLURM [8] to manage and to schedule an individual cluster, and those instances are often connected together by *separate* job management software, like MOAB [9], PBS Pro [10], and LSF [11]. While simple, the increasing interplay between various classes of compute resources across the center renders this paradigm often inflexible and ineffective. For example, this paradigm cannot effectively schedule applications that utilize site-wide shared resources such as file systems. Without scheduling file I/O-intensive jobs to both compute resources and file systems, overlapping I/O bursts coming from only a handful of unrelated jobs can disrupt the entire center [6], [4]. In addition, with such an inflexible and shallow hierarchy, dynamically imposing complex resource constraints at various levels at the center is becoming increasingly intractable.

To address these challenges, we need a new management paradigm that is capable of scalably managing all of the resources (e.g., compute, storage, and visualization) at a center together **under one common RJMS framework**, (co-)scheduling jobs to these various types of resources, and dynamically enforcing global and local resource constraints. The only way to achieve this with scalability in terms of the total number of resources as well as jobs is that the RJMS system must facilitate management and scheduler parallelism [12], [13] and further provide this parallelism through hierarchical, multilevel management and scheduling schemes with support for arbitrary numbers of levels. The latter also requires capabilities for customization or specialization (in terms of policy, access control, etc) at any level of this dynamic hierarchy based on user requests.

In this paper, we present Flux, a new open-source RJMS framework that implements this new paradigm. We will describe its design and run-time architecture and show how it will be capable of providing the necessary RJMS capabilities for next-generation systems and centers. To demonstrate its feasibility, we have prototyped and evaluated two of its core run-time components: the communication framework termed the Comms Message Broker (CMB) and Key Value Store (KVS) to hold Flux state. CMB and KVS together represent a

novel back-bone overlay network for Flux, which can replace all the redundant and independent daemon infrastructures that currently exist in a typical cluster.

This paper makes the following contributions:

- We describe, based on experience in one of the world's largest HPC centers, the emerging resource and job management challenges that demand a new paradigm;
- We present the design and run-time architecture of Flux, a novel RJMS system that embodies this new paradigm;
- We evaluate two of the core run-time components of Flux to demonstrate the feasibility of our approach.

The results of our performance evaluation and performance models show that both KVS and CMB provide strong and predictable scaling properties and provide a scalable foundation for the Flux architecture.

## II. A New Management Paradigm for HPC

HPC centers typically provide a wide range of platforms on which scientific applications perform computations. The RJMS system is responsible for efficiently delivering compute cycles of these platforms to multiple users who must submit jobs to run their applications. Thus, the RJMS matches the users' job request with the available resources and provides efficient functions for building, submitting, launching, and monitoring jobs [1]. Typically, an HPC center runs an RJMS instance [8] to manage and schedule an individual system (e.g., cluster) and additionally employs separate grid software to tie these instances together.

However, several growing trends present major challenges to this current management paradigm. First, systems sited at one center are growing larger and types of resources are becoming increasingly diverse [1]. Second, interplays between various classes of resources (e.g., between compute clusters and a global file system) are becoming more complicated and disruptive [6], [4]. Third, resource constraints are also becoming increasingly complex in a multidimensional, dynamic, and hierarchical fashion [14] (e.g., dynamic power capping at the level of systems, compute racks, and/or nodes). Further, the need for code-development tools and their requirements on tool launching and job access impose challenging requirements on the RJMS [3], [4], [5], [6], [7]. Finally, the workloads themselves are becoming diverse, dynamic, and large, and are moving away from individual monolithic jobs. Instead, ensembles of jobs, e.g., for Uncertainty Quantification or Scalebriding Applications, are becoming increasingly commonplace.

To address the issues effectively, large HPC centers demand a new resource and job management paradigm. The new paradigm must increase its purview and scalably manage resources across the entire center. Perhaps more importantly, it must be implemented under one common RJMS framework so that its schedulers can make use of its full resource representations. Only this allows centers to (co-)schedule jobs effectively to various types of resources and to provide much richer provenance on jobs (e.g., correlation between a user-level error and other system activities). The new paradigm, however, creates a series of challenges a new RJMS must overcome:

*Challenge 1: Multidimensional Scaling* — The new paradigm requires that the RJMS can impose complex, multidimensional resource bounds at any scale, from the center-wide level, down to the level of individual processes, and enable the most efficient execution and scheduling of workloads within these bounds. This imposes unprecedented scale challenges in multiple dimensions, supporting extreme scalability, addressing noise as concurrency increases, and managing a drastically increased amount of run-time information that must be monitored, traced, and stored. The new paradigm must efficiently handle increased scale in numbers of resources as well as jobs and other dimensions of RJMS data.

*Challenge 2: Diverse workloads* — HPC applications are known to have disparate performance limiting factors. This requires the new paradigm to have a rich resource model, including the representation for diverse types of resources such as file systems, networks, visualization hardware, and heterogeneous compute engines. With a richer resource model, the RJMS will be capable of imposing complex, multidimensional resource bounds, as opposed to the simplistic traditional resource model that is fundamentally based on a flat list of nodes, and allowing it to allocate resources tailored to the disparate limiting factors of HPC applications.

*Challenge 3: Dynamic Workloads* — Resource allocations must be elastic, i.e., resource allocations must be able to grow and shrink dynamically. This is necessary to support HPC applications with different phases with disparate performance-limiting factors. Different resource types have different elastic properties (e.g., power is a much more elastic resource than compute nodes) restricting decision points and allocation granularity. One consequence of this is that the RJMS must support multiple levels of elasticity as a function of dynamically changing performance limiters as well as their limiting resource types—e.g., rigid vs. moldable vs. malleable scheduling [15] against different workload and resource types.

*Challenge 4: Productivity* — The new paradigm must address increasing complexities in code development and system administration by facilitating the creation of more effective diagnostic and analysis tools. For example, it must provide basic, scalable monitoring and communication primitives at the job level that can be leveraged by tools. It will encourage a richer, stronger tool ecosystem. Better tools will lead to higher productivity for all stakeholders, including end users.

*System Challenges* — In addition to these main requirements that come from the user side of the RJMS, we also face internal challenges that need to be hidden from the user. For example, in a global model, the risk of higher downtime costs arises. If the RJMS is inadequately designed, a downtime could negatively impact the availability of a large portion of the center's resources. Thus, it must be tolerant of hardware and software faults and failures with no single point of failure and also support live software upgrades. Other challenges include security, integration risk, and backwards compatibility.

This series of challenges strongly motivates a specific management and scheduling scheme: the new RJMS must hierarchically and dynamically manage and schedule the resources under one common software framework. The divide-

and-conquer approach will then allow the RJMS to scale to massive amounts of resources sited at a large center. Further, the hierarchical, multilevel job scheduling will then facilitate scheduler parallelism [12], [13], and this will allow the RJMS to scale to massive numbers of jobs scheduled across the center. In this scheme, higher-level schedulers must allow a site to impose site-wide policies, contracts, and constraints while lower-level schedulers should allow efficient use of any subsets of resources in accordance with workload types. Finally, the RJMS must be capable of dynamically supporting arbitrarily deep levels in this management and scheduler hierarchy with an ability to impose different constraints at each level.

## III. Conceptual design of Flux

In this section, we present Flux, a novel open-source RJMS framework that targets the paradigm discussed above. In the following we will describe its fundamental design concepts.

*Unified Job Model:* In the traditional paradigm, a job is simply defined to be a resource allocation. Flux, however, abstracts this notion to an independent RJMS instance that can either be used to run a single application or that can run its own job management services, which then can recursively accept and schedule (sub-)jobs and provide its own services to them. In the following we will refer to this simply as a Flux job or just a job. To provide the necessary flexibility, each RJMS instance allows specialized service plugins to be instantiated so that the resources managed by the Flux job can be used differently than other resources (in terms of security, scheduling policy, or resource constraints). This model forms the foundation for hierarchical, multilevel resource management and job scheduling with resource subset specialization.

*Job Hierarchy Model:* To scale the new paradigm to the entire HPC center, we must avoid a centralized approach. Instead, Flux exploits the hierarchical resource management and job scheduling discussed above and organizes itself in a tree-based hierarchy of Flux jobs. Several guiding principles throughout the job hierarchy strike a balance between the management responsibility of a parent job and the delegation and empowerment of a child job:

- Parent bounding rule: the parent job grants and confines the resource allocation of all of its children.
- Child empowerment rule: within the bounds set by the parent, the child job is delegated the ownership of the allocation and becomes solely responsible for most efficient uses of the resources.
- Parental consent rule: the child job asks its parent when it wants to grow or shrink the resource allocation, and it is up to the parent to grant the request.

This model has many advantages for scalability of both resource management and job scheduling. The independent and specialized Flux service instance of a job becomes only responsible for managing its direct children jobs, which would be only a small fraction of the total number of jobs at the center. As sibling jobs run simultaneously, their independent Flux instances will perform concurrent management services.

For example, for job scheduling, this model enables Flux to exploit scheduling parallelism [12], [13]. A parent scheduler schedules at coarse granularity over a large collection of resources and leases different resource subsets to its children schedulers. At the same time, this will enable Flux to specialize the scheduling behaviors on subsets of resources without having to introduce a complex global scheduling policy into the centralized, monolithic scheduler.

*Generalized Resource Model:* In the traditional paradigm, compute resources are modeled primarily as a collection of compute nodes. But this is a simplistic perspective ill-suited for the new paradigm. Today's applications are diverse with disparate limiting performance factors beyond floating point computation. Further, computing centers are increasingly concerned about managing new resource types such as power and shared persistent storage. Flux therefore introduces a generalized resource model that is extensible and covers any kind of resource and its relationships. This enables scheduling decisions based on many types of resources.

*Multilevel Resource Elasticity Model:* As our applications and their programming models are becoming increasingly dynamic, the new paradigm demands an elasticity model where an existing resource allocation can grow and shrink, depending on the current needs of applications and/or the HPC center. Flux supports the elasticity model within our job hierarchy framework above: a child job sends a grow or shrink request to its parent, which can be aggregated up the job hierarchy until all requisite constraints are known for this request. Also, combining this with the generalized resource model, the elasticity can be expressed for many resources such as power and file I/O bandwidth.

*Common Scalable Communication Infrastructure Model:* To maintain scalability both within and across jobs, Flux provides a common scalable communication framework within each job. When a Flux job is created, a secure, scalable overlay network with common communication service is established across its allocated nodes. Except for the root-level job, the existing communication session of the parent job assists the child job with rapid creation of its own session.

A communication session is only aware of its parent and child and passes the limited set of control information through its communication channel. Thus, this model provides highly scalable communication to management services within a Flux job, while limiting communications between jobs, addressing both the multidimensional scale as well as security issues. Further, this per-job backbone communication network supports well-known bootstrap interfaces for distributed programs including many MPI implementations as well as run-time tools. This provides tightly integrated support for the development and use of scalable run-time tools.

## IV. Flux Run-time System

To explore the feasibility of the Flux design, we implemented prototypes of the two key components in the Flux run-time environment.

### A. Communication Message Broker

A communication framework supports our hierarchical job model by establishing a *comms session* to contain each Flux instance and provide a foundation for the distributed

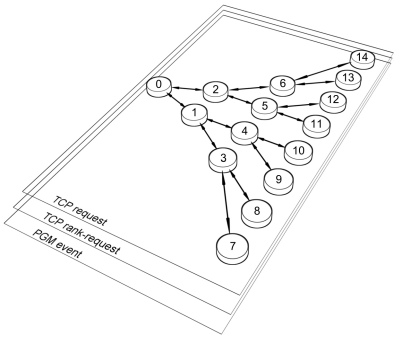



Fig. 1: A comms session

components upon which Flux is built. As this framework must follow our common communication infrastructure model, it enables secure, scalable communication within a comms session, limits communication between sessions, and allows new comms sessions to be created, resized, destroyed, and monitored by existing ones in a parent-child relationship. The framework is persistent for the life of a Flux job and is shared among Flux services, tools, and application run-times.

We have built a prototype of the framework using the ØMQ [16] messaging library, which provides the ability to pass messages securely over multiple transports, including TCP, UNIX domain IPC, shared memory, and Pragmatic General Multicast (PGM) [17]. Its socket-like API abstracts common messaging patterns such as *request-reply*, *publish-subscribe*, and *push-pull*.

Our prototype consists of a distributed Comms Message Broker (CMB) daemon that runs on each node of a comms session, interconnected using three persistent overlay network planes: a PGM *publish-subscribe* bus for events and synchronization heartbeats; a TCP *request-response* tree for scalable RPCs, barriers, and reductions; and a secondary TCP *request-response* overlay with configurable topology for rank-addressed RPCs.

The comms session wire-up is depicted in Figure 1. Each message plane implements reliable, in-order message delivery, and can self-heal when interior nodes fail. Although a binary RPC/reduction tree is pictured, the tree shape is configurable. A design for comprehensive fault tolerance, including root node failure, is a near-term project activity.

The CMB allows us to experiment with loosely coupled distributed services that share this message routing framework. The various service components of Flux have been implemented as *comms modules*, plugins which are loaded into the CMB address space and pass messages over shared memory. Comms modules currently exist for the components listed in Table I. Each embodies an active topic for study and experimentation.

In addition to comms modules, external programs communicate with the CMB over a UNIX domain socket. A `flux` utility wraps command line access to about two dozen modular Flux sub-commands, and a custom PMI [18] library allows MPI run-times to access the Flux KVS and collective barrier modules over this transport.

All CMB messages have a uniform, multi-part message format consisting of at least a *header frame* and a *JSON [19] frame*. The header frame identifies the message recipient using a hierarchical name space. For example, a message sent to

| Plugin | Description |
|---|---|
| hb | A periodic heartbeat event multicast across the comms session synchronizes background activity to reduce scheduling jitter. |
| live | Each tree node receives heartbeat-synchronized *hello* messages from its children. After a configurable number of missed messages, a liveness event is issued for a dead child. |
| log | Log messages are reduced and filtered before being placed in a log file at the session root. A circular debug buffer provides log context in response to a fault event. |
| mon | Lua scripts stored in the KVS activate heartbeat-synchronized sampling. Samples are reduced and stored in the KVS. |
| group | Flux groups define and manage collection of processes that can participate in collective operations. |
| barrier | Collective barriers provide synchronization across Flux groups. |
| kvs | A distributed key-value store provides a scalable multi-purpose data store for Flux and other tools operating within the comms session. |
| wrexec | Remote processes can be launched in bulk, monitored, receive signals, and have standard I/O captured in the KVS. |
| resrc | Resources are enumerated in the KVS and allocated when the scheduler runs an application. |

TABLE I: Prototyped Comms Modules

*kvs.put* is routed to the *kvs* comms module, and internally to its handler for *put*. The free-form JSON frame contains payload parsed by the addressed comms module.

RPC requests are routed "upstream" in the tree network to the first comms module that matches it, possibly traversing CMB nodes. RPC responses are routed back through the same set of hops, in reverse. A comms module may thus be loaded at a configurable tree depth to tune its level of distribution or to conserve node resources for application workloads toward the leaves. The tree topology of the RPC overlay network permits data reductions to be performed by aggregating and retransmitting upstream requests between instances of a comms module.

Alternatively, an RPC may be addressed to a specific CMB rank using a separate overlay, currently utilizing a ring topology which allows ranks to be trivially reached without routing tables. The main use of this addressing mode is in tools for debugging the system, where the high latency of a ring is preferable to additional complexity.

### B. Distributed Key-Value Store

Key-Value Stores (KVS) have become ubiquitous building blocks in large-scale Internet services but have been underutilized in HPC [20]. We determined that a KVS would be a essential building block for our new system. The Flux KVS is implemented as a comms module that utilizes the request-response and event overlay networks. It provides a general purpose data store used by other Flux components, and supports the distributed caching and synchronization needed for parallel data exchanges, such as required for MPI bootstrap.

Our current prototype stores JSON values under a hierarchical key space with a single master node and multiple caching slaves. The weak consistency of our slave caches has the following properties, using Vogels' taxonomy [21].

- *Causal consistency*: If process A communicates with process B that it has updated a data item (passing a *store version* in that message), a subsequent access by process B will return the updated value.
- *Read-your-writes consistency*: A process having updated a data item, never accesses an older value.
- *Monotonic read consistency*: If a process has seen a particular value for an object, any subsequent accesses will never return previous values.

We achieve these properties with a simple design based on hash trees and content-addressable storage, borrowing ideas from ZFS [22] and git. JSON objects are placed in a content-addressable *object store*, hashed by their SHA1 digests. Hierarchical key names are broken up into path components that reference directories. A directory is an object that maps a list of names to other objects by their SHA1 reference. An external root directory SHA1 reference points to the root directory object. For example, if the SHA1 root reference is `1c002dde...`, and we have stored `a.b.c = 42`, we would look it up as follows:

1) load root directory from `1c002dde...`, find `a` is at `3f2243ef...`.
2) load `a` from `3f2243ef...`, find `b` is at `023e9b2d...`.
3) load `b` from `023e9b2d...`, find `c` is at `7ff234a8...`.
4) load `c` from `7ff234a8...`, and return it (42).

An important property of this structure is that any update results in a new SHA1 root reference. Continuing the example, to update `a.b.c = 43`, we:

1) store 43 to `62302aff...`.
2) update `b` to associate `c` with `62302aff...`, and store `b` to `8fe9b2c3...`.
3) update `a` to associate `b` with `8fe9b2c3...`, and store `a` to `aacc76b4...`.
4) update root to associate `a` with `aacc76b4...`, and store root to `033fbe92...`.
5) the new root reference is `033fbe92`.

All updates are applied first on the master node at the root of the CMB tree, which then publishes a new root reference as a CMB event. Slaves keep consistent with the master by switching their root reference in response to this event, so that all new look-ups must begin at the new root directory. Objects missing from the slave object cache during a look-up are faulted in from their CMB-tree parent, recursing up the tree until the request can be fulfilled. Unused slave object cache entries are expired after a period of disuse to save memory.

The CMB event overlay network guarantees ordered delivery, which gives us monotonic read consistency for free. We achieve read-your-writes consistency by returning the new root reference in response to a commit request and applying it before returning to the caller. We avoid racing with the event update and potentially breaking monotonic read consistency by versioning the root references and ensuring they are never applied out of order. We achieve causal consistency by allowing this version number to be read after an update, and by providing another call to wait for this root version or greater on another node before accessing the value.

The KVS API includes classes of functions for putting, committing, and getting KVS objects. First, `kvs_put (key, val)` writes *val* to the object store asynchronously in a write-back mode through the tree of slave caches. The (*key, SHA1*) tuple is cached locally pending commit.

`kvs_commit ()` synchronously flushes (*key, SHA1*) tuples and any still-dirty objects to the master. On the master, it then processes the set of tuples, creating new directory objects as described above, finally arriving at a new root SHA1. It then updates the root reference session-wide with a multicast event. Since both new and old objects coexist in the caches, the switch from old to new root is atomic. `kvs_get_version ()` and `kvs_wait_version ()` are available for causal consistency as described above. `kvs_fence ()` commits for a group of processes collectively through the internal use of a collective barrier.

`kvs_get (key)` recursively looks up the key starting with the current root reference, faults in any missing objects through the tree of slave caches, and returns the terminal object. `kvs_watch (key, callback)` is a `get` variant which registers a callback to be triggered whenever the value of *key* changes. It accomplishes this by internally performing a `get` on the watched value in response to each root update, comparing the new and old values, and calling the callback if they are different. Due to our hash-tree organization, a watched directory changes if keys under it at any path depth change.

## V. Results

In the following, we evaluate the performance and scalability of our CMB and KVS prototypes, using a dedicated test called KVS Access Patterns (KAP). KAP allows us to stress both the KVS abstraction as well as the underlying CMB communication. It models KVS access patterns through various interactions between KVS writers and readers. Writers are called producers; readers consumers. In essence, KAP allows a configurable number of producers to write key-value objects into our KVS and a configurable number of consumers to read these objects after ensuring the consistent KVS state.

In addition to producer and/or consumer counts, KAP provides a range of parameters that can affect performance, including the value size (of key-value objects), the number of objects to put, the number of objects to get, objects access patterns (through different striding), and synchronization primitives used for consistency. KAP consists of four phases: setup, producer, synchronization and consumer phases. During the setup phase, tester processes are launched into a set of nodes in which a CMB session had been established. They are assigned ranks such that consecutive rank processes are distributed to consecutive nodes. The rank processes determine their roles based on the command line arguments (either producer or consumer) and issue a Flux collective barrier to begin to play their roles simultaneously.

Next, each producer calls the specified number of `kvs_puts` of an object of the specified value size. For each call, the producers use unique keys, but the values can be configured to be either unique or redundant. Once this is done, all of the producers and consumers enter the synchronization phase in which they participate in a consistency protocol. They use KVS synchronization primitives such as `kvs_fence` and `kvs_wait_version`. Finally, during the consumer phase, consumers read these key-value objects by calling `kvs_gets`. KAP provides options to emulate various read access patterns.

### A. Experimental Setup

We ran all of our experiments on two Linux clusters installed at Lawrence Livermore National Laboratory (LLNL), named Zin and Cab. Each compute node of these clusters has

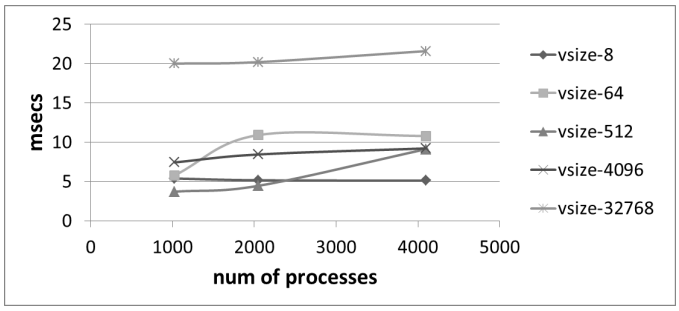



Fig. 2: Max latency of producer phase

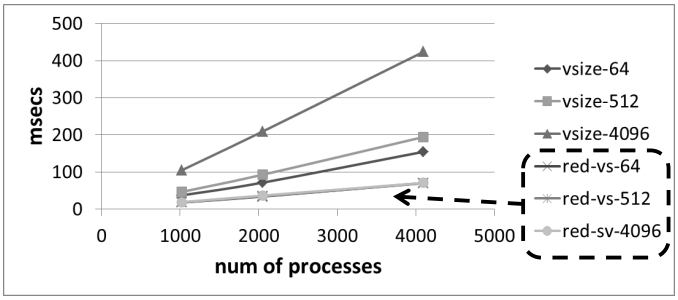



Fig. 3: Max latency of synchronization phase

2 sockets and 32 GB of RAM. Each socket is populated with an 8-core 2.6 GHz Intel Xeon E5-2670 processor, resulting in 16 cores per node. Nodes are connected by a Qlogic Infiniband QDR interconnect. The largest allocation our batch system allows for our tests is 512 nodes (8,192 cores).

We ran KAP with varying arguments to its parameters in batch mode and collected performance metrics. Due to the huge parameter space, however, we limited our experiments to only a subset of the parameter set by using a sampling strategy.

Specifically, we ran our KAP tests at 64, 128, 256 and 512 compute nodes, and always fully populated each node with 16 processes, each acting as consumer or producer or both. We varied the consumer or producer count while fixing the other at the total number of cores. We also tested the value size at 8, 32, 128, 512, 2048, 8192, and 32,768 bytes, and the key-value object access count of each consumer from 1 to the total process count.

Further, we evaluated the performance impact of how key-value objects are organized in KVS by either storing all of the objects into a single KVS directory or distributing them into multiple directories of at most 128 objects each. Finally, we studied the performance implications of redundancy in values by either configuring producers to generate unique or redundant values across them. For simplicity, we fixed the comms session topology as a binary tree and used only `kvs_fence` for synchronization.

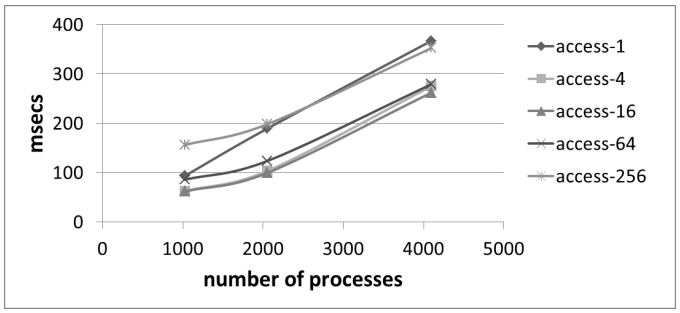
### B. Performance Results and Analysis

Of tens of thousands of our sampling runs, we find that the fully populated cases—i.e., both producer and consumer counts become equal to the total process count—are most revealing. In particular, we carefully analyze the maximum latency of each of the main phases of KAP for these cases because this metric represents the critical path of the performance of many HPC process-management services. For example, distributed HPC software would use KVS operations in a coordinated fashion to exchange connection information among processes during its bootstrapping phase as shown in LIBI [23] and PMI [18]. Unless *all* of the distributed processes complete their KVS operations, their communication fabric cannot be established.

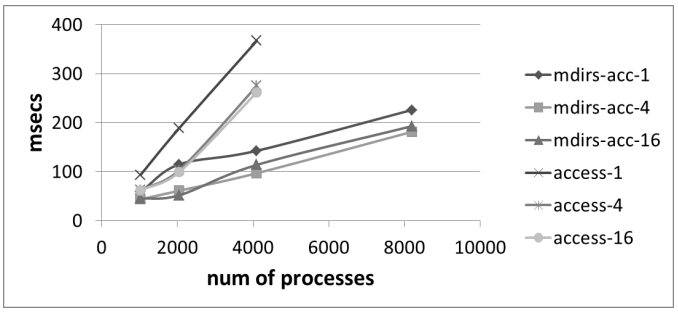
Figure 2 shows the maximum latency of the producer phase. Essentially, these plots indicate how well `kvs_put` scales as we increase the number of producers. Each plot represents different value sizes—e.g., `vsize-8` refers to value size being 8 bytes. As shown in this graph, the `kvs_put` simply performs and scales well. This matches our expectations because objects are cached in write-back mode at `kvs_put` time and flushed to the master at the next consistency event.

Moving on to the synchronization phase, Figure 3 shows how `kvs_fence` scales as the number of producers increase. As with the producer latency, each plot represents different value sizes under two different value types: unique values vs. redundant values. The most revealing observation is that fence scalability depends on the level of redundancy in key-value objects that had previously been put in. Figure 3 shows the improvements made when redundant values are used. Label `red-vsize-k` is the same maximum latency metric as `vsize-k` except that redundant values had been used.

We theorize that in the unique-value case, fence would perform linearly with respect to the number of producers because these values are simply being *concatenated* while being sent up the tree, and that it would perform logarithmically for the redundant-value case because redundant values are *reduced* while being sent up the tree. However, our data shows that the redundant-value case falls short of logarithmic scaling, and we find that this is because while values are reduced, the *(key, SHA1)* tuples referring to them are still concatenated.




(a) With single-directory layout




(b) Improvements with multiple directories

Fig. 4: Max latency of consumer phase (value size: 8 bytes)

Next, Figure 4 shows the maximum latency of the consumer phase. These results show how `kvs_get` scales as we increase the number of consumers. Each plot represents the latency when each consumer reads different numbers of objects, e.g., the `access-4` plot represents each consumer reading 4 distinct objects. While these figures show only the performance of reading objects 8 bytes in size, we observe that scalability trends are similar at different value sizes.

Figure 4(a) shows the maximum latency of `kvs_get` when the target keys are all stored in a single KVS directory object. The latency is quite high and also increases linearly as we increase the number of consumers. It appears that the poor performance and scaling behavior are attributed to the fact that our slave caches store only full objects, and the small objects being consumed in the test cannot be retrieved without faulting in the entire directory object containing them, through the tree of CMB slave cache instances.

With our access pattern where $G$ objects are read collectively by $C$ consumers, and the time to replicate $G$ objects in a single slave cache from its CMB-tree parent is given by $T(G)$, the maximum consumer latency is given by

$$log_2(C) \times T(G). \tag{1}$$

Thus, the max latency increase for every doubling of consumers is

$$\frac{log_2(2C) \times T(2G)}{log_2(C) \times T(G)}. \tag{2}$$

Generally, this would approach 2, as we continue to increase the number of consumers, and our data match with this model.

We can improve this behavior by storing objects across multiple directories. Slave caches can then operate at a finer granularity, and the quantity of data that must be retrieved to satisfy consumer requests will tend, depending on how requests stride directories, to be proportional to the quantity of data requested. This is especially true toward the leaves of the CMB tree where the slave caches service a decreasing subset of consumers. Figure 4(b) shows the improvements when objects are spread into directories of 128 objects each. Label `mdir-acc-k` refers to the same access pattern as `access-k` except the objects are stored across multiple directories.

When objects are spread across directories and the amount of data faulted into slave caches decreases as a function of tree levels, the latency can be modeled as a geometric series. For example, at tree height $h$, the latency would be

$$T(G) + T(\frac{G}{2}) + T(\frac{G}{4}) + ... + T(\frac{G}{2^h}), \tag{3}$$

where each term represents the latency of replication per level, and the sum approaches $2T(G)$. Thus, its improvement over the single directory scheme is on the order of

$$\frac{log_2(C) \times T(G)}{2T(G)} = \frac{1}{2} log_2(C). \tag{4}$$

The improvements would be linearly greater as we increase the scale and our measurements agree with this.

While these results suggest a promising avenue for reworking the KVS internal object layout for performance, we note that such a scheme alone would fall short of reaching an extremely large scale. Our model suggests that the latency will grow linearly when $G$ grows with the scale. For example, if $G$ doubles every time the number of consumers is doubled, our geometric series model predicts the latency will also double according to

$$\frac{2T(2G)}{2T(G)}. \tag{5}$$

With the current scheme, the only way to gain true logarithmic scaling is when $G$ stays constant regardless of scale.

## VI. Related Work

Flux seeks to change the paradigm in which large HPC centers should manage, model, schedule, and allocate resources. A strong body of research exists in each of these areas including production solutions such as SLURM [8], LSF [11], Moab [9], PBS Pro [10], LoadLeveler [24] and Condor [25]. Flux is distinguished from these approaches, as it provides a new paradigm that can address emerging resource and job management challenges with a center-wide purview under one common software framework.

In other areas like cloud and grid computing, the need for exploiting scheduler parallelism including two-level scheduling recently emerged. Google's Omega [12] exploits a parallel scheduler architecture whereby multiple schedulers concurrently access the shared resource state with an optimistic concurrency model. Efforts such as Apache Mesos [13] and many emerging grid schedulers [26], [27] take advantage of two-level scheduling strategies. However, these approaches are neither optimized for HPC workloads nor well suited for large HPC centers. Scales of large HPC centers like those at national laboratories demand a deeper and more dynamic levels in the resource management and scheduler hierarchy with an ability to impose constraints at various levels in this hierarchy.

HPC trends increasingly motivate scalable KVS implementations. Wang et al. proposed a distributed KVS as the basis for HPC services to encapsulate complexity of distributed services [20]. They further evaluate replacing the centralized controller in SLURM [8] with a distributed controller [28] built on ZHT [29]. While existing KVS work takes an incremental approach of improving scalability of a traditional RJMS paradigm with new scalable services, ours are built specifically to support the new paradigm. We also evaluated Redis [30] and twemproxy [31] as part of our early KVS investigation. However, their design points are not optimized for HPC workloads which often feature synchrony and coordination.

Finally, much research exists in the area of tree-based overlay network (TBON). They include MRNet [32] and COBO [7], and CMB can be considered to be a TBON. But unlike user-level TBONs, ours must support system-level activities, and this requires us to address distinct research topics such as support for multiple user-level networks (which actually include other user-level overlay networks), security, low noise and fault tolerance.

## VII. Conclusion

Large HPC centers are increasingly facing multifaceted resource and job management challenges that if not properly met, will result in significant losses. We present a new management

paradigm and its incarnation, Flux, that can effectively address these challenges in a single software framework, while making site-wide operations more tractable.

We have validated two of the key run-time components responsible for communication, and our results show that our run-time infrastructure is most scalable when information exchange patterns themselves are also scalable, which suggests two significant directions. For one, we must carefully design the data exchange patterns among distributed components of run-time elements including Flux's own management services. In particular, to achieve extreme scalability, each component must avoid accessing a global view. Secondly, we must also continue to push the scalability envelope of our infrastructure, in particular in the KVS. We plan to address the latter by distributing the KVS master itself.

Ultimately, we are designing Flux to enable developers at the operating system and run-time levels to leverage the RJMS data stores and services in new and powerful ways, and we expect that it will position HPC centers to cope with diverse, massively large-scale resources.

## VIII. Acknowledgments

The authors acknowledge the contributions of Matthieu Hautreux of Commissariat a l'Energie Atomique et aux Energies Alternatives and Jon Bringhurst, formerly of Los Alamos National Laboratory, who provided substantial and insightful feedback on the Flux vision document and in subsequent discussions. We are also indebted to Jeff Long, Chris Morrone, Ned Bass, Mark Gary, Kim Cupps, Pam Hamilton and Scott Futral of LLNL who provided detailed feedback for our initial project review. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

## References

[1] Y. Georgiou, "Contributions for resource and job management in high performance computing," Ph.D. dissertation, Joseph Fourier University, 2010.

[2] Lawrence Livermore National Laboratory, "Advanced Simulation and Computing Sequoia," https://asc.llnl.gov/computing_resources/sequoia.

[3] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*, March 2007, pp. 1–10.

[4] W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf, "Massively parallel loading," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 389–398. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465020

[5] D. H. Ahn, G. L. Lee, G. Gopalakrishnan, Z. Rakamarić, M. Schulz, and I. Laguna, "Overcoming extreme-scale reproducibility challenges through a unified, targeted, and multilevel toolset," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, ser. SE-HPCCSE '13. New York, NY, USA: ACM, 2013, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/2532352.2532357

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[7] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Overcoming scalability challenges for tool daemon launching," in *Proceedings of the 37th International Conference on Parallel Processing*, 2008, pp. 578–585.

[8] M. A. Jette, A. B. Yoo, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[9] "Moab workload manager version 7.2.6," Adapting Computing, Administrator Guide, 2014.

[10] Altair, "PBS professional," 2014. [Online]. Available: http://www.pbsworks.com/Product.aspx?id=1

[11] IBM, "IBM Platform LSF," January 2014. [Online]. Available: http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/prod%ucts/lsf/index.html

[12] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 351–364. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465386

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[14] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 173–182. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465009

[15] D. G. Feitelson and L. Rudolph, "Towards convergence in job schedulers for parallel supercomputers," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, ser. IPPS '96. London, UK, UK: Springer-Verlag, 1996, pp. 1–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=646377.689507

[16] P. Hintjens, "ØMQ - the guide, updated for version 3.2," Retrieved Nov. 5, 2012 from http://zguide.zeromq.org/page:all.

[17] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano, "PGM Reliable Transport Protocol Specification," RFC 3208 (Experimental), Internet Engineering Task Force, Dec. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3208.txt

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 31–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894122.1894127

[19] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627 (Informational), Internet Engineering Task Force, Jul. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[20] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503239

[21] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[22] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," Tech. Rep., 2003.

[23] J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee, B. R. De Supinski, M. P. Legendre, B. P. Miller, and M. Schulz, "LIBI: A framework for bootstrapping extreme scale software systems," *Parallel Comput.*, vol. 39, no. 3, pp. 167–176, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2012.09.003

[24] IBM, "Loadleveler." [Online]. Available: http://publib.boulder.ibm.com/clresctr/windows/public/llbooks.html

[25] M. Litzkow, M. Livny, and M. Mutka, "Condor-a hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988, pp. 104–111.

[26] M. Pasquali, R. Baraglia, G. Capannini, L. Ricci, and D. Laforenza, "A multi-level scheduler for batch jobs on grids," *The Journal of Supercomputing*, vol. 57, no. 1, pp. 81–98, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11227-011-0571-y

[27] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, ser. CCGRID '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 776–783.

[28] X. Zhou, H. Chen, K. Wang, M. Lang, and I. Raicu, "Exploring distributed resource allocation techniques in the slurm job management system," Illinois Institute of Technology, Technical Report, 2013.

[29] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 775–787. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.110

[30] S. Sanfilippo, "redis key-value store web site," Retrieved Nov. 9, 2012 from http://redis.io/.

[31] M. Rajashekhar, "twemproxy: a fast, lightweight proxy for memcached and redis," Retrieved Jan. 22, 2014 from https://github.com/twitter/twemproxy.

[32] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 21–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050172