Exploring the Design Tradeoffs for Extreme-Scale High-Performance Computing System Software

Ke Wang, Abhishek Kulkarni, Michael Lang, Dorian Arnold, and Ioan Raicu

Abstract—Owing to the extreme parallelism and the high component failure rates of tomorrow's exascale, high-performance computing (HPC) system software will need to be scalable, failure-resistant, and adaptive for sustained system operation and full system utilizations. Many of the existing HPC system software are still designed around a centralized server paradigm and hence are susceptible to scaling issues and single points of failure. In this article, we explore the design tradeoffs for scalable system software at extreme scales. We propose a general system software taxonomy by deconstructing common HPC system software into their basic components. The taxonomy helps us reason about system software as follows: (1) it gives us a systematic way to architect scalable system software by decomposing them into their basic components; (2) it allows us to categorize system software based on the features of these components, and finally (3) it suggests the configuration space to consider for design evaluation via simulations or real implementations. Further, we evaluate different design choices of a representative system software, i.e. key-value store, through simulations up to millions of nodes. Finally, we show evaluation results of two distributed system software, Slurm++ (a distributed HPC resource manager) and MATRIX (a distributed task execution framework), both developed based on insights from this work. We envision that the results in this article help to lay the foundations of developing next-generation HPC system software for extreme scales.

Index Terms—Distributed systems, High-performance computing, Key-value stores, Simulation, Systems and Software

1 Introduction

System software is a collection of important middleware services that offer to upper-lay applications integrated views and control capabilities of the underlying hardware components. Generally system software allows applications full and efficient hardware utilization. A typical system software stack includes (from the bottom up) operating systems (OS), runtime systems, compilers, and libraries [1]. Technological trends indicate that exascale high-performance computing (HPC) systems will have billion-way parallelism [2], and each node will have about three orders of magnitude more intra-node parallelism than that of the node of today's petascale systems [4]. Exascale systems will pose fundamental challenges of managing parallelism, locality, power, resilience, and scalability [3][5][6].

Current HPC system software designs focus on optimizing the inter-node parallelism by maximizing the bandwidth and minimizing the latency of the interconnection networks but suffer from the lack of scalable solutions to expose the intra-node parallelism. New loosely-coupled programming models (e.g. many-task computing [38], over-decomposition [7], and MPI + OpenMP [8]) are helping to address intra-node parallelism for exascale systems. These programming models place a high demand on system software for scalability, fault-tolerance and adaptivity. However, many of the existing HPC system software are still designed around a centralized server paradigm and, hence, are susceptible to scaling issues and single points of failure. Such concerns suggest a move towards fundamentally scalable distributed system software designs – a move further motivated by the growing amount of data (and metadata) that servers need to maintain in a scalable, reliable, and consistent manner.

The exascale community has been exploring research directions that address exascale system software challenges, such as lightweight OS and kernels (e.g. ZeptoOS [9], Kitten [10]); asynchronous and loosely coupled runtime systems (e.g. Charm++ [11], Legion [12], HPX [13], STAPL [14], and Swift [15]); load balanced and locality-aware execution and scheduling models (e.g. MATRIX [23][25], ParallelX [16], and ARMI [17]); automatic and auto-tuning compilers (e.g. ROSE [18], SLEEC [19]). The general collections of HPC system software are those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding, and various runtime systems for programming models and communication libraries. As HPC systems approach exascale, the basic design principles of scalable and fault-tolerant system architectures need to be investigated for HPC system software implementations. Instead of exploring the design choices of each system software at every stack level individually and in an ad hoc fashion, this work aims to

- *K. Wang is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: kwang22@hawk.iit.edu.*
- *A. Kulkarni is with the Department of Computer Science, Indiana University, Bloomington, IN 47405. E-mail: adkulkar@cs.indiana.edu.*
- *M. Lang is with the Los Alamos National Laboratory, Los Alamos, NM 87544. E-mail: mlang@lanl.gov.*
- *D. Arnold is with the Department of Computer Science, University of New Mexico, Albuquerque, NM 87131. E-mail: darnold@cs.unm.edu.*
- *I. Raicu is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: iraicu@cs.iit.edu.*

xxxx-xxxx/0x/$xx.00 © 2015 IEEE Published by the IEEE Computer Society

develop a general framework that allows for systematic explorations of the design space of HPC system software and to evaluate the impacts of different design choices.

In this article, the questions we intend to answer are: ***what are the scalabilities of different system software architectures (centralized, hierarchical, distributed); and at what scales and levels of reliability and consistency does distributed design outweigh the extra complexity and overhead of centralized and hierarchical designs.***

To answer the questions, we devise a general taxonomy that classifies system software based on basic components, so as to identify their performance and scaling limits. By identifying the common basic components and focusing on designing these core components, we will enable faster prototyping and development of new system software. We then motivate key-value stores (KVS) as a building block for HPC system software at extreme scales, and then using KVS as a case study, we explore design tradeoffs of system software. Via simulation, we explore the scalability of each system architecture and quantify the overheads in supporting reliability at extreme scales. Finally, we evaluate two system software (a distributed HPC resource manager, SLURM++ [22], and a distributed task execution framework, MATRIX [23][24][25]), which are developed based on the insights from this work. We believe the work presented in this article lays the foundations for the development of the next-generation, extreme-scale HPC system software.

This article extends our previous work [26] that motivated KVS as a building block for extreme-scale HPC system software and evaluated different KVS designs. The contributions of the previous work were: (1) a taxonomy for classifying KVS; (2) a simulation tool to explore KVS design choices for large-scale system software; and (3) an evaluation of KVS design choices at extreme scales using both synthetic workloads and real workload traces.

The extension covers the aspects of both depth and broadness of scope. For broadness, we focus on general HPC system software instead of just KVS. For depth, we add a hierarchical architecture in the comparison with the centralized and distributed ones. We also evaluate more system software that apply KVS as distributed metadata management to demonstrate more extensively that KVS is a fundamental block for extreme-scale system software.

The new contributions of this article are as follows:

1. We devise a comprehensive taxonomy by deconstructing system software into their core components. The taxonomy helps us reason about general system software as follows: (1) it gives a systematic way to decompose system software into their basic components; (2) it allows one to categorize system software based on the features of these components, and finally, (3) it suggests the configuration spaces to consider for evaluating system designs via simulation or real implementation.

2. We conduct an inclusive evaluation of different system architectures (centralized, hierarchical, and distributed) under various design choices, such as different replication, recovery, and consistency models.

3. We offer empirical evaluations of other system software that use KVS for metadata management. This supports proposal of using KVS as a building block for HPC system software at extreme scales.

The rest of this article is organized as follows. Section 2 motivates KVS as a building block and identifies the centralized architecture's bottleneck for HPC system software at extreme scales; Section 3 presents the taxonomy and shows how the taxonomy can help to classify existing system software; Section 4 details the KVS simulation design and implementation; Section 5 evaluates different architectures through simulations up to millions of nodes, and offers the evaluation of two system software that apply KVS as distributed metadata management; Section 6 discusses other related research; Section 7 presents our conclusions and opportunities for future work.

2 KEY-VALUE STORES IN HPC

2.1 Building Blocks for HPC

We motivate that KVS is a building block for HPC system software at extreme scales. The HPC system software, which we generally target, are those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding, and various runtime systems for programming models and communication libraries [29][30][31][32]. For extreme-scale HPC systems, these system software all need to operate on large volumes of data in a scalable, resilient and consistent manner. We observe that such system software commonly and naturally comprise of data-access patterns amenable to the NoSQL abstraction, a lightweight data storage and retrieval paradigm that admits weaker consistency models than traditional relational databases.

These requirements are consistent with those of large-scale distributed data centers, such as, Amazon, Facebook, LinkedIn and Twitter. In these commercial enterprises, NoSQL data stores – Distributed Key-Value Stores (KVS) in particular – have been used successfully [33][34][35] in deploying *software as a service* (SaaS). We assert that by taking the particular needs of HPC system into account, we can use KVS for HPC system software to help resolve many scalability, robustness, and consistency issues.

By encapsulating distributed system complexities in the KVS, we can simplify HPC system software designs and implementations. Giving some examples as follows: For resource management, KVS can be used to maintain necessary job and node status information. For monitoring, KVS can be used to maintain system activity logs. For I/O forwarding in file systems, KVS can be used to maintain file metadata, including access authority and modification sequences. In job start-up, KVS can be used to disseminate configuration and initialization data amongst composite tool or application processes (an example of this is under development in the MRNet project [32]). Application developers from Sandia National Laboratory [36] are targeting KVS to support local checkpoint/restart protocols. Additionally, we have used KVS to implement several system software, such as a many-task computing (MTC) task execution [37][38][39][40] framework – MATRIX [23][24][25], where KVS is used to store the task metadata information, and a fuse-based distributed file

system, FusionFS [41], where the KVS is used to track file metadata.

2.2 Centralized Architecture's Bottleneck

HPC system software designed around the centralized architecture suffer from limited scalability, high likelihood of non-recoverable failures and other inefficiencies. To validate this, we assess the performance and resilience of a centralized file-backed KVS.

We implement a KVS prototype. Each request (*put*, *get*, and *delete*) was turned into a corresponding file system operation (*write*, *read* and *remove*, respectively) by the server. A request with a 16-byte payload is consist of a (*key*, *value*) pair. We run the prototype on a 128-node machine with AMD 2GHz Dual-Core Opteron and 4 GB of memory per node. Compute nodes are connected with Gigabit Ethernet. At every second boundary, the throughput attained by the server is measured to determine the maximum throughput during operation.

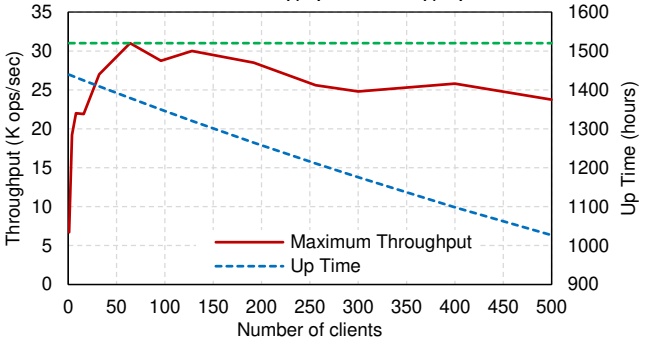


Fig 1: Performance and Resilience of Centralized KVS

Fig 1 shows the peak throughput is achieved at 64 clients with the configuration of one client per node. As multiple clients per node are spawned, the throughput decreases due to network contention. At relatively modest scales, centralized server shows significant performance degradation due to contention.

To measure the reliability of the centralized server, we set the failure rate of the server to be dependent on the number of clients it is serving due to the increasing loads on the server. Considering an exponential distribution of server failures, the relationship between the server's up time and the number of clients is: $up\ time = Ae^{\lambda n}$, where A is the up time with zero client, λ is the failure rate, and n is the number of clients. Assuming a 2-month up time with zero client ($A = 1440$ hours), and a 1-month (i.e. 720 hours) up time with 1024 clients of a single server, we show the trend of the server up time with respect to the number of clients (dotted blue line) in Fig 1. The reliability decreases as the scale of the system increases.

At exascale, the above results would be amplified to pose serious operability concerns. *While not surprising results, these results motivate alternative distributed architectures that support scalability, reliability and consistency in a holistic manner.* These issues can be addressed by identifying the core components required by system software, such as a global naming system, an abstract KVS, a decentralized architecture, and a scalable, resilient overlay network.

3 HPC System Software Taxonomy

In contrast to the traditional HPC system software that are tightly coupled for synchronized workloads, SaaS developed for the Cloud domain is designed for loosely asynchronous embarrassingly parallel workloads in distributed systems with wide area networks. As HPC systems are approaching exascale, the HPC system software will need to be more asynchronous and loosely coupled to expose the ever-growing intra-node parallelism and hide latency. To be able to reason about general HPC system software at exascale, we devise a taxonomy by breaking system software down into various core components that can be composed into a full system software. We introduce the taxonomy, through which, we then categorize a set of system software.

A system software can be primarily characterized by its **service model**, **data layout model**, **network model**, **recovery model**, and **consistency model**. These components are explained in detail as follows:

(I) Service Model describes system software functionality, architecture, and the roles of the software's composite entities. Other properties such as *atomicity*, *consistency*, *isolation*, *durability* (ACID) [27], *availability*, *partition-tolerance* etc. also are expressed as parts of the service model. These characteristics define the overall behavior of the system software and the constraints it imposes on the other models. A transient data aggregation tool, a centralized job scheduler, a resource manager with a single failover, a parallel file system are some examples of the service model.

(II) Data Layout Model defines the system software data distribution. In a centralized model, a single server is responsible for maintaining all the data. Alternatively, the data can be partitioned among distributed servers with varying levels of replication, such as partitioned (no replication), mirrored (full replication), and overlapped (partial replication).

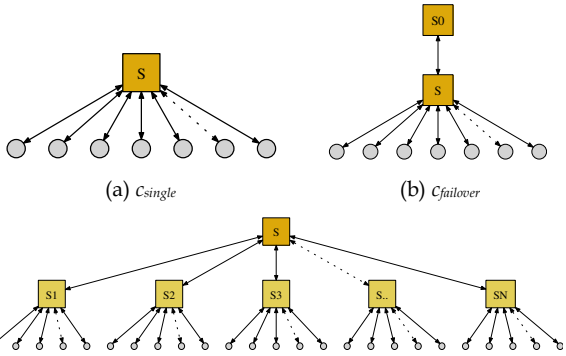
(III) Network Model dictates how system software components are connected. In a distributed network, servers can form structured overlays – rings, binomial, k-ary, n-cube, radix trees; complete, binomial graphs; or unstructured overlay – random graphs. The system software could be further differentiated based on deterministic or non-deterministic information routing in the overlay network. While some overlay networks imply a complete membership set (e.g. fully-connected), others assume a partial membership set (e.g. binomial graphs).

(IV) Recovery Model describes how system software deals with server failures with minimum manual intervention. The most common methods include fail-over, checkpoint-restart, and roll-forward. Triple modular redundancy and erasure coding [20] are additional ways to deal with server failures and ensure data integrity. The recovery model can either be self-contained, such as recovery via logs from persistent storage, or require communication with others to retrieve replicated data.

(V) Consistency Model pertains to how rapidly data changes in a distributed system are propagated and kept coherent. Depending on the data layout model and the corresponding level of replication, system software may

employ different levels of consistency. The level of consistency is a tradeoff between the server's response time and how tolerant clients are to stale data. It can also compound the complexity of recovery under failures. Servers could employ weak, strong, or eventual consistency depending on the importance of the data.

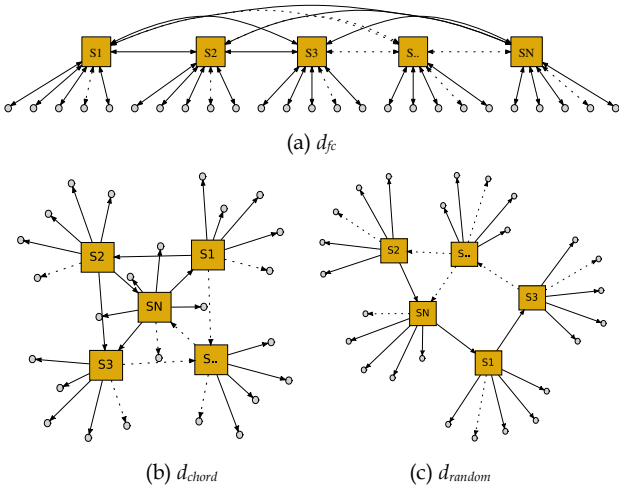
By combining specific instances of these components, we can define a system architecture of system software. Fig 2 and Fig 3 depict some specific system architectures derived from the taxonomy. For instance, c_{tree} is a system architecture with a centralized data layout model and a tree-based hierarchical overlay network; d_{fc} architecture has a distributed data layout model with a fully-connected overlay network, whereas d_{chord} architecture has a distributed data layout model and a Chord overlay network [28] with partial membership. Recovery and consistency models are not depicted, but would need to be identified to define a complete service architecture.



(a) c_{single} (b) $c_{failover}$

(c) c_{tree}

Fig 2: Centralized system architecture

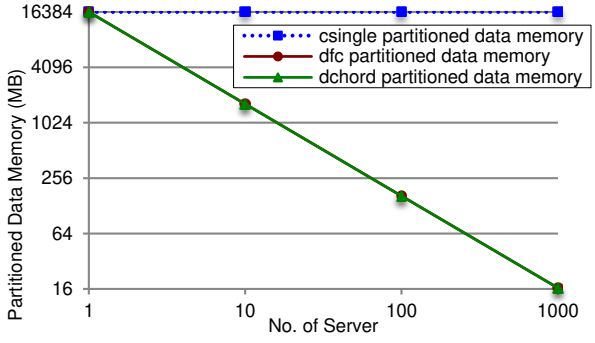


(a) d_{fc}

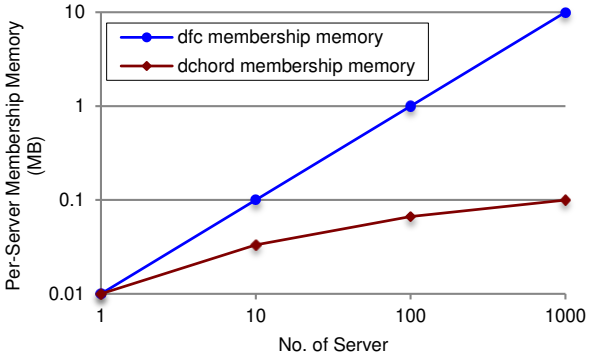
(b) d_{chord} (c) d_{random}

Fig 3: Distributed system architecture

Looking into the memory requirements of these architectures allows deriving observations analytically. Fig 4 (a) shows the per-server memory requirement of the client data for different architectures, assuming 16GB client data. A single server must have the memory capacity to hold all the data, where the d_{fc} and d_{chord} architectures partition the data evenly across many servers. Fig 4 (b) illustrates the per-server memory requirements to store the server membership information, assuming each server identification is 10KB. This is trivial for a single server. For d_{fc}, it grows linearly with the number of severs, while for d_{chord}, the relationship is logarithm.



(a) Data memory per server



(b) Membership memory per server

Fig 4: Memory requirements for different architectures

To demonstrate how common HPC system software would fit into the taxonomy, we have classified, at a high-level, some representative system software in Table 1.

4 Key-value Stores Simulation

Having motivated KVS as a building block for extreme-scale HPC system software, using the taxonomy we narrow the parameter space and focus on the major KVS components. Then we can use simulation to evaluate the design spaces for any specific KVS applications before any implementation. Additionally, we can create modular KVS components that allow the easy creation of extreme-scale system software. This section presents the design and implementation details of a KVS simulator. The simulator allows us to explore all the system architectures, namely c_{single}, c_{tree}, d_{fc} and d_{chord}. Here we assume a centralized data layout model for c_{single} and c_{tree}, and a distributed data layout model for d_{fc} and d_{chord}. The simulator is extendable to other network and data layout models. The architectures can be configured with N-way replication for the recovery model and either eventual or strong consistency for the consistency model. The conclusions that we will draw from KVS simulations can be generalized to other system software, such as job schedulers, resource managers, I/O forwarding, monitoring, and file systems.

Table 1: Representative system services categorized through the taxonomy

System Software	*Service Model*	*Data Layout Model*	*Network Model*	*Recovery Model*	*Consistency Model*
Charm++	Runtime System	Distributed	Hierarchical	N-way Replication	Strong
Legion	Runtime System	Distributed	Hierarchical	None	Strong
STAPL	Runtime System	Distributed	Nested/Hierarchical	N-Way Replication	Strong
HPX	Runtime System	Distributed	Fully-connected	N-Way Replication	Strong
SLURM	Resource Manager	Replicated	Centralized	Fail-Over	Strong
SLURM++	Resource Manager	Distributed	Fully-connected	Fail-Over	Strong
MATRIX	Task Scheduler	Distributed	Fully-connected	None	Strong
OpenSM	Fabric Manager	Replicated	Centralized	Fail-Over	Strong
MRNet	Data Aggregation	Centralized	Hierarchical	None	None
Lilith	Data Distribution	Replicated	Hierarchical	Fail-Over	Strong
Yggdrasil	Data Aggregation	Replicated	Hierarchical	Fail-Over	Strong
IOFSL	I/O Forwarding	Centralized	Hierarchical	None	None
Riak	Key-value store	Distributed	Partially-connected	N-way Replication	Strong and Eventual
FusionFS	File System	Distributed	Fully-connected	N-way Replication	Strong and Eventual

4.1 Simulator Overview

Simulations are conducted up to exascale levels with tens of thousands of nodes (each one has tens to hundreds of thousands threads of execution) running millions of clients and thousands of servers. The clients are simulated as compute daemons that communicate with the servers to store data and system states. The millions of clients are spread out as millions of compute daemon processes over all the highly parallel compute nodes. Furthermore, the number of clients and servers are configurable.

The data records stored in the servers are (*key*, *value*) pairs; the *key* uniquely identifies the record and the *value* is the actual data object. By hashing the *key* through some hashing function (e.g. modular) over all the servers, the client knows the exact server that is storing the data. Servers are modeled to maintain two queues: a **communication queue** for sending and receiving messages and a **processing queue** for handling incoming requests that operate on the local data. Requests regarding other servers' data cannot be handled locally and are forwarded to the corresponding servers.

4.2 Data Layout and Network Models

The data layout and network models supported are: centralized data server (c_{single}), centralized data server with aggregation servers in a tree overlay (c_{tree}), distributed data servers with fully connected overlay (d_{fc}), distributed data servers with partial connected overlay (d_{chord}).

For c_{single} and c_{tree}, all data is stored in a single server. The main difference is that c_{tree} has a layer of aggregation servers to whom the client submits requests. Currently, the aggregation servers only gather requests.

For d_{fc} and d_{chord}, the *key* space along with the associated data *value* is evenly partitioned among all the servers ensuring a perfect load balancing. In d_{fc}, data is hashed to the server in an interleaved way (*key* modular the server id), while in d_{chord}, consistent hashing [42] is the method for distributing data. The servers in d_{fc} have global knowledge of all servers, while in d_{chord}, each server has only partial knowledge of the other servers; specifically this is logarithm of the total number of servers with base 2 and is kept in a table referred to as the *finger table* in each server.

4.3 Recovery Model

The recovery model defines how a server recovers its state and how it rejoins the system after a failure. This includes how a recovered server recovers its data and how to update the replica information of other servers that are affected due to the recovery. The first replica of a failed server is notified by an external mechanism (**EM**) [28] (e.g. a monitoring system software that knows the status of all servers) when the primary server recovers. Then the first replica sends all the replicated data (including the data of the recovering server and of other servers for which the recovering server acts as a replica) to the recovered server. The recovery is done once the server acknowledges that it has received all data.

We implement a replication model in the simulator for the purpose of handling failures. In c_{single} and c_{tree}, one or more failovers are added; while in d_{fc} and d_{chord}, each server replicates its data in the consecutive servers (servers have consecutive id numbers from 0 to server count - 1). Failure events complicate server replication model. When a server fails, the first replica sends the failed server's data to an additional server to ensure that there are enough replicas. In addition, all the servers that replicate data on the failed server would also send their data to one more server. The clients can tolerate server failures by identifying the replicas of a server as consecutive servers.

Our simulator implements different policies for the clients to handle server failures, such as *timeouts*, *multi-trial*, and *round-robin*. For example, in the *timeouts* policy, a client would wait a certain time for the server to respond. If the server doesn't respond after the timeout, the client then turns to the next replica. In addition, our simulator has the ability to handle communication failures by relying on the EM. The EM monitors the status of all the servers by issuing periodic heart-beat messages. When a link failure of a server happens, the EM detects it according to the failed heart-beat message and then notifies the affected clients, which then direct requests to the next replica.

4.4 Consistency Model

Our simulator implements two consistency models: strong consistency and eventual consistency [21].

4.4.1 Strong Consistency

In strong consistency, updates are made with atomicity guarantee so that no two replicas may store different values for the same *key* at any given time. A client sends requests to a dedicated server (**primary replica**). The *get* requests are processed and returned back immediately. The *put* requests are first processed locally and then sent to the replicas; the **primary replica** waits for an acknowledgement from each other replica before it responds back to the client. When a server recovers from failure, before getting back all its data, it caches all the requests directed to it. In addition, the first replica (notified by the **EM**) of the newly recovered server migrates all pending *put* requests, which should have been served by the recovered server, to the recovered server. This ensures that only the **primary replica** processes *put* requests at any time while there may be more than one replicas processing *get* requests.

4.4.2 Eventual Consistency

In eventual consistency, given a sufficiently long period of time over which no further updates are sent, all updates will propagate and all the replicas will be consistent eventually, although different replicas may have different versions of data of the same *key* at a given time. After a client finds the correct server, it sends requests to a random replica (called the **coordinator)**. This is to model inconsistent updates of the same *key* and also to achieve load balancing, among all the replicas. There are three key parameters to the consistency mechanism: the number of replicas–**N**, the number of replicas that must participate in a quorum for a successful *get* request–**R**, and the number of replicas that must participate in a quorum for a successful *put* request-**W**. We satisfy **R+W>N** to guarantee "read our writes" [21]. Similar to Dynamo [33] and Voldemort [35], we use **vector clock** to track different data versions and detect conflicts. A **vector clock** is a <serverId, counter> pair for each *key* in each server. It specifies how many updates have been processed by the server for a *key*. If all counters in a **vector clock** V1 are no larger than all corresponding ones in a **vector clock** V2, then V1 precedes V2, and can be replaced by V2. If V1 overlaps with V2, then there is a conflict.

For a *get* request, the **coordinator** reads the value locally, sends the request to other replicas, and waits for <value, vector clock> responses. When a replica receives a *get* request, it first checks the corresponding **vector clock**. If it precedes the **coordinator**'s, then the replica responds with success. Otherwise, the replica responds failure, along with its <value, vector clock> pair. The **coordinator** waits for **R**−1 successful responses, and returns all the versions of data to the client who is responsible for reconciliation (according to an application-specific rule such as "largest value wins") and writing back the reconciled version.

For a *put* request, the **coordinator** generates a new **vector clock** by incrementing the counter of the current one by 1, and writes the new version locally. Then the **coordinator** sends the request, along with the new **vector clock** to other replicas for quorum. If the new **vector clock** is preceded by a replica's, the replica accepts the update and responds success; otherwise, responds failure. If at least **W**−1 replicas respond success, the *put* request is considered successful.

4.5 KVS Simulator Implementation Details

After evaluating several simulation frameworks such as OMNET++ [43], OverSim [44], SimPy [45], PeerSim [46], we chose to develop the simulator on top of PeerSim because of its support for extreme scalability and dynamicity. We use the discrete-event simulation (DES) [47] engine of PeerSim. Every behavior in the system is converted to an event and tagged with an occurrence time. All the events are inserted in a global event queue that is sorted based on the event occurrence time. In every iteration, the simulation engine fetches the first event and executes the corresponding actions, which may result in following events. The simulation terminates when the queue is exhausted.

The simulator is developed in Java (built on top of PeerSim) and has about 10,000 lines of code. The input to the simulation is a configuration file, which specifies the system architecture, and the system parameters.

5 Evaluation

Our evaluation aims to give insights into the design spaces of HPC system software through KVS simulations, and to show the capabilities of our simulator in exposing costs inherent in design choices. We evaluate the overheads of different architectures as we vary the major components defined in section 3. We present results by incrementally adding complex features such as replication, failure/recovery, and consistency, so that we can measure the individual contributions to the overheads due to supporting these distributed features.

The simulations are run on a single node; the largest amount of memory required for any of the simulations is 25GB and the longest running time is 40 minutes (millions of clients, thousands of servers, and tens of millions of requests). Given our lightweight simulator, we can explore an extremely large scale range.

Our simulator has been validated against two real key-value stores, ZHT [48] and Voldemort [35], within moderate scales, 8K nodes for ZHT and 500 nodes for Voldemort. The simulator reported a relatively small average difference of 4.38% comparing with ZHT and of 10.71% comparing with Voldemort. The validation details can be found in our prior work [26].

One important metric used in our evaluation is *efficiency*. The *efficiency* is the percentage ratio of the ideal running time to the actual running time of a given workload. The ideal running time is calculated by accounting to merely request processing time and assuming zero communication overheads. The *efficiency* quantifies the average utilization of the system. Higher *efficiency* numbers indicate less communication overheads.

5.1 Architecture Comparisons

We compare different architectures for the basic scenario (no replication, failure/recovery or consistency models) with synthetic workloads to investigate the tradeoffs between these system architectures at increasingly large scales. In the synthetic workload, each client submits 10 requests with 5 *get* operations and 5 *put* operations on the *key* space of 128-bit (generated with a uniform random distribution), and each request message is 10KB.

5.1.1 c_{single} vs. c_{tree}

Fig 5 shows the comparison between c_{single} and c_{tree}. We see that before 16 clients, c_{tree} performs worse than c_{single} due to that the small gather size (at most 16) is insufficient to make up the additional latency of the extra communication hop. Between 32 (1 aggregation server) to 16K clients (16 aggregation servers with each one managing 1K clients), c_{tree} performs better than c_{single} because of the larger gather sizes (32 to 1K). After 32K clients, the individual performance is degrading, the relative performance gap is decreasing and finally disappearing. This is because the per-request processing time is getting larger when the number of clients increases due to contentions, which renders that the communication overhead is negligible.

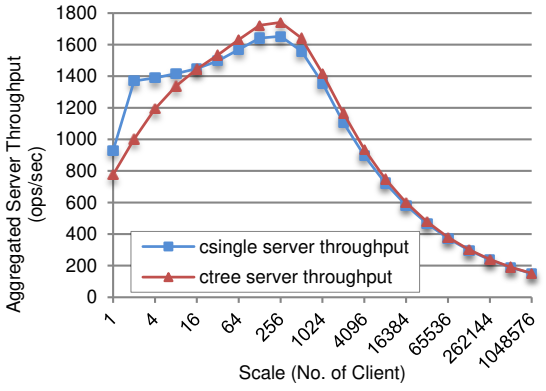


Fig 5: Throughput of c_{single} vs c_{tree}

To model the server contention due to the increasing number of clients, we run a prototype of a centralized KVS (c_{single}) implementation up to 1K nodes and apply a polynomial regression on the processing time with respect to the number of clients with the base of 500ms. Within 1K nodes, the changing of processing time is shown in Table 2. The 1K-client processing time (637 ms) is used in d_{fc} and d_{chord} as each server manages 1K clients. For c_{single} and c_{tree}, Beyond 1K nodes, we increase the processing time linearly with respect to the client count.

In Fig 5, the values after 1K clients are linear models. There could be other models (e.g. logarithm, polynomial, exponential) between processing time and the number of clients depending on the server implementation (e.g. multi-threading, event-driven, etc). We only use the calibrated values up to 1K clients. We show the results after 1K clients merely to point out that **there is a severe server contention in a single server at large scales, leading to poor scalability of the centralized architecture**.

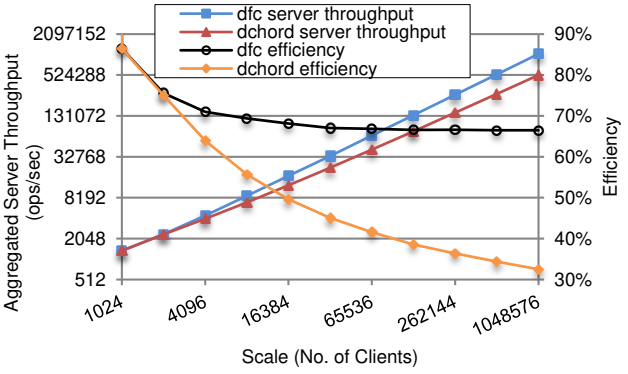
Table 2: Processing time as a function of number of clients

Number of Clients	1	2	4	8	16	32	64	128	256	512	1K
Processing Time (ms)	613	611	608	601	588	567	537	509	505	541	637

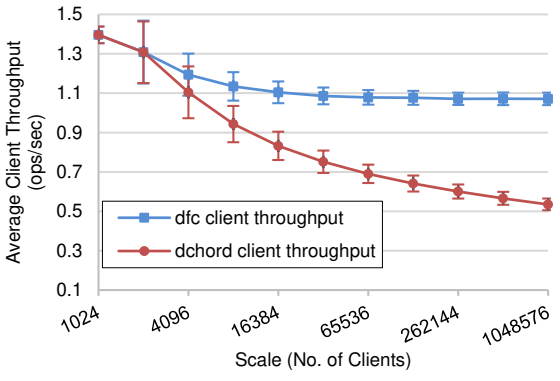
5.1.2 d_{fc} vs. d_{chord}

The comparison between d_{fc} and d_{chord} is shown in Fig 6. Each server is configured to manage 1K clients. With d_{fc}, we observe that the server throughput almost scales linearly with respect to the server count, and the efficiency has a fairly constant value (67%) at extreme scales, meaning that d_{fc} has great scalability. With d_{chord}, we see slightly less throughput as we scale up, and the efficiency decreases smoothly (Fig 6(a)). This is due to the additional routing required by d_{chord} to satisfy requests: one-hop maximum for d_{fc} and $logN$ hops for d_{chord}.

We show the average per-client throughput for both d_{fc} and d_{chord} in Fig 6(b). Up to 1M clients, d_{fc} is about twice as fast as d_{chord} from the client's perspective. From the error bars, we see that d_{chord} has higher deviation of the per-client throughput than that of d_{fc}. This is again due to the extra hops required to find the correct server in d_{chord}.



(a) Server throughput and efficiency



(b) Average throughput per client

Fig 6: d_{fc} and d_{chord} performance comparison

The conclusion is that at the base case, the partial connectivity of d_{chord} results in latency as high as twice as that of the full connectivity of d_{fc}, due to the extra routing.

5.2 Server Failure Effect with Replication

This section explores the overhead of failure events when a server is configured to keep updated replicas for resilience. We choose the *multi-trial* policy: the clients resend the failed requests to the primary server several times (an input parameter) before turning to the next replica.

Fig 7 displays the *efficiency* comparison between the base d_{fc} and d_{fc} configured with failure events and replication, and between the base d_{chord} and d_{chord} configured with failure events and replication, respectively. We use 3 replicas, set the failure rate to be 5 failure events per minute, and apply a strong consistency model. As seen in Fig 7, both d_{fc} and d_{chord} have significant *efficiency* degradation when failures and replication are enabled (blue solid line vs blue dotted line, red solid line vs red dotted line). The performance degradation of d_{fc} is more severe than that of d_{chord} - 44% (67% to 23%) for d_{fc} vs. 17% (32% to 15%) for d_{chord}. We explain the reasons with the help of Table 3.

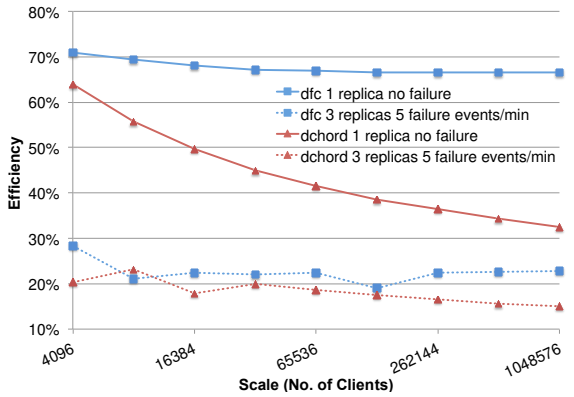


Fig 7: Server failure effect with replication

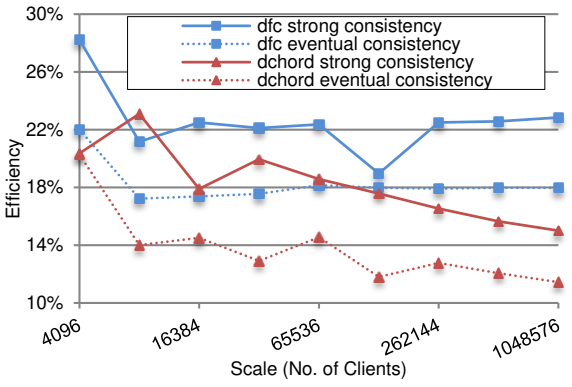


Fig 8: Strong consistency and eventual consistency

Table 3 lists the message count of each property (process request, failure, strong consistency) for both d_{fc} and d_{chord}. We see that at extreme scales, the request-process message count (dominant factor) does not increase much when turning on failures and replicas for both d_{fc} and d_{chord}. The message count of failure event is negligible, and of strong consistency increases significantly at the same rate for both d_{fc} and d_{chord}. However, these added messages account for 1/3(20M/60M) for d_{fc}, while less than 1/8 (20M/170M) for d_{chord}. Due to the high request-process message count in d_{chord}, the overhead of d_{fc} seems more severe. The replication overhead is costly, which indicates that tuning a system software to the appropriate number of replicas will have a large impact on performance.

5.3 Strong and Eventual Consistency

We compare the overheads of consistency models in this section. We enable failures with 5 failure events per minute and use 3 replicas. Like Dynamo [33], for eventual consistency, we configure (**N**, **R**, **W**) to be (3, 2, 2).

Fig 8 shows the *efficiency* results of both d_{fc} and d_{chord}. We see that eventual consistency has larger overhead than strong consistency. From strong to eventual consistency, *efficiency* reduces by 4.5% for d_{fc} and 3% for d_{chord} at extreme scales. We also list the number of messages for request-process, failure events, and consistency models in Table 4. We observe that the request-process message count doesn't vary much for both d_{fc} and d_{chord}. However, for consistency messages, eventual consistency introduces about as twice (41M/21M) the number of messages as that of strong consistency. This is because in eventual consistency, each request would be forwarded to all **N**=3 replicas and the server waits for **R**=2 and **W**=2 successful acknowledgments. With strong consistency, just the *put* requests would be forwarded to all other replicas. Eventual consistency gives faster response times to the clients but with larger cost of communication overhead.

5.4 KVS Applicability to HPC System Software

In this section, we show that KVS can be used as building

Table 3: Message count for d_{fc}, d_{chord} with and without failure and replica (F&R)

	request-process message count				failure message count		strong consistency message count	
# Clients	d_{fc}	d_{chord}	d_{fc} (F&R)	d_{chord} (F&R)	d_{fc} (F&R)	d_{chord} (F&R)	d_{fc} (F&R)	d_{chord} (F&R)
4096	143.4K	185.0K	312.4K	246.2K	33	4.5K	217.7K	87.1K
8192	307.3K	491.1K	404.1K	596.2K	42	445	175.4K	170.9K
16384	634.8K	1.2M	726.1K	1.5M	66	28.6K	336.5K	377.1K
32768	1.3M	2.8M	1.4M	3.0M	114	712	665.4K	662.0K
65536	2.6M	6.4M	2.7M	6.6M	210	590	1.3M	1.3M
131072	5.2M	14.3M	5.3M	14.5M	402	888	2.6M	2.6M
262144	10.5M	31.3M	10.6M	31.7M	786	996	5.3M	5.2M
524288	21.0M	67.9M	21.1M	68.4M	1.6K	1.1K	10.5M	10.5M
1048576	41.9M	146.6M	42.0M	147.1M	3.1K	1.3K	21.0M	21.0M

Table 4: Message count of strong consistency (sc) and eventual consistency (ec) for d_{fc} and d_{chord}

	process message count				failure message count				consistency message count			
	sc		ec		Sc		ec		sc		ec	
# Clients	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}
4096	312.4K	246.2K	141.5K	211.4K	30	4.6K	30	360	217.7K	87.1K	167.2K	164.5K
8192	404.1K	596.2K	391.9K	682.7K	40	450	50	590	175.4K	170.8K	340.2K	328.2K
16384	726.1K	1.5M	733.1K	1.5M	67	28.6K	90	23.7K	336.5K	377.1K	668.2K	655.4K
32768	1.4M	3.0M	1.4M	3.1M	110	710	150	830	665.4K	661.9K	1.3M	1.3M
65536	2.7M	6.7M	2.7M	6.6M	210	590	210	770	1.3M	1.3M	2.6M	2.6M
131072	5.3M	14.5M	5.3M	14.8M	400	890	530	1.1K	2.6M	2.6M	5.3M	5.3M
524288	21.1M	68.4M	21.0M	68.7M	1.6K	1.1K	2.1K	1.4K	10.5M	10.5M	21.0M	21.0M
1048576	42.0M	147.1M	42.0M	148.0M	3.1K	1.3K	4.1K	1.6K	21.0M	21.0M	42.0M	42.0M

block for developing HPC system software. First, we conduct simulations with three workloads, which were obtained from real traces of three system software: job launch using SLURM, monitoring by Linux Syslog, and I/O forwarding using the FusionFS [41] distributed file system. Then, we evaluate two distributed system software that use a KVS (i.e. ZHT [48]) for distributed state management, namely an HPC resource manager, SLURM++ [22], and a MTC task scheduler, MATRIX [23].

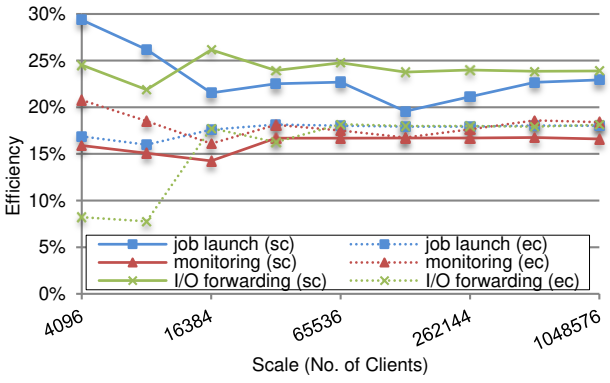
5.4.1 Simulation with Real Workload Traces

We run simulations with three workloads obtained from typical HPC system software, listed as follows:

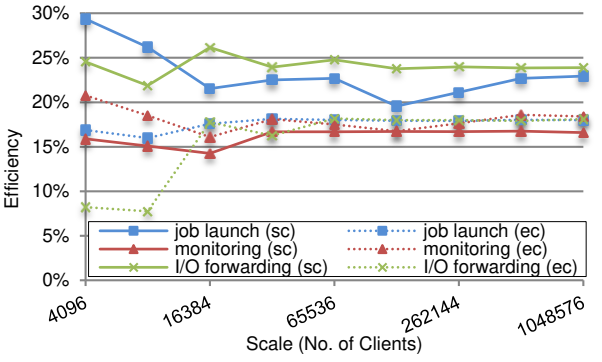
1. **Job Launch:** this workload is obtained from monitoring the messages between the server and client during an MPI job launch in SLURM resource manager. Though the job launch is not implemented in a distributed fashion, the messages should be representative regardless of the server structure, and in turn drive the communications between the distributed servers. The workload is characterized with the controlling messages of the slurmctld (*get*) and the results returning from the slurmds (*put*).
2. **Monitoring:** we get this workload from a 1600-node cluster's syslog data. The data is categorized by message-type (denoting the *key* space) and count (denoting the frequency of each message). This distribution is used to generate the workload that is completely *put* dominated.
3. **I/O Forwarding:** We generate this workload by running the FusionFS distributed file system. The client creates 100 files and operates (*reads* or *writes* with 50% probability) on each file once. We collect the logs of the ZHT metadata servers that are integrated in FusionFS.

We extend these workloads to make them large enough for exascale systems. For job launch and I/O forwarding, we repeat the workloads several times until reaching 10M requests, and the *key* of each request is generated with uniform random distribution (URD) within 64-bit *key* space. The monitoring workload has 77 message types with each one having a different probability. We generate 10M *put* requests; the *key* is generated based on the probability distribution of the message types and is mapped to 64-bit *key* space. We point out that these extensions reflect some important properties of each workload, even though cannot reflect every details: the job launch and I/O forwarding workloads reflect the time serialization property and the monitoring workload reflects the probability distribution of all obtained messages.

We run these workloads in our simulator, and present the *efficiency* results for d_{fc} and d_{chord} with both strong and eventual consistency, in Fig 9. We see that for job launch and I/O forwarding, eventual consistency performs worse than strong consistency. This is because both workloads have almost URD for request type and the *key*. For monitoring workload, eventual consistency does better because all requests are *put* type. The strong consistency requires acknowledgments from all the other **N**-1 replicas, while the eventual consistency just requires **W**-1 acknowledgments. Another fact is that the monitoring workload has the lowest *efficiency* because the *key* space is not uniformly generated, resulting in poor load balancing.



(a) d_{fc}



(b) d_{chord}

Fig 9: d_{fc} and d_{chord} with real workloads

The above results demonstrate that our KVS framework can simulate various system software as long as the workloads could be mirrored to *put* or *get* requests, which is true for the HPC system software we have investigated.

5.4.2 SLURM++ Distributed Resource Manager

The majority use cases of traditional big machines are running large-scale (e.g. full scale or at least jobs with a large percentage of the machine) tightly coupled HPC applications with long durations (e.g. days to weeks). In addition, strict policies are set to limit the number of concurrent job submissions per client, and the job sizes are suggested to be large (e.g. 512 nodes on the IBM BG/P machine in ANL). These constraints make great efforts to guarantee the resource allocation of high-priority large jobs, although it has been criticized that these policies often lead to low utilizations. In this scenario, a centralized resource manager with a single job scheduling component performs adequately as there are only limited number of decisions of resource allocation and scheduling that need to be made at a time.

As the exascale machines will have about one order of magnitude more nodes with each one having up to three orders of magnitude more parallelism (making up billion-way parallelism), we argue that besides the traditional large-scale HPC jobs, many orders of magnitude more asynchronous jobs with a wide distribution of job sizes and durations should be supported concurrently, in order to maximize the system utilization. Because only a small number of applications can scale up to exascale requiring full-scale parallelism, most applications will be decomposed with the high-order low-order methods, which have many small-scale coordinated ensemble jobs with

shorter durations. In addition, as the compute node will have much higher parallelism at exascale, it is important to support asynchronous parallel MTC workloads that are fine-grained in both job size (e.g. per-core task) and durations (e.g. from sub-second to hours). Furthermore, we hope that the scheduling policies will be changed to allow many users to submit more jobs of various requirements of resources concurrently. The mixture of applications and the ever-growing number of job submissions will pose significant scalability and reliability challenges on the resource manager and job scheduler.

One potential solution is to partition the whole machine and enable distributed resource management and job scheduling in multiple partitions. In designing the next-generation resource manager for exascale machines, we have developed a prototype of distributed resource manager, SLURM++, based on the SLURM centralized resource manager, combined with the fully distributed (d_{fc}) KVS, ZHT. SLURM++ comprises of multiple controllers, and each one manages a partition of SLURM daemons (slurmd), in contrast to SLURM's centralized architecture (a single controller, slurmctld manages all the slurmds). The controllers use ZHT to keep the free node list in local partition and to resolve resource contentions (via the atomic *compare and swap* [49] operation of ZHT).

To achieve dynamic resource balancing, we develop a random resource stealing technique. When launching a job, a controller first checks the local free nodes. If local partition has enough free nodes, the controller directly allocates them; otherwise, it queries ZHT for other partitions from which it will steal resources. The technique keeps stealing nodes from random controllers until the job allocation is satisfied. For systems with heterogeneous interconnections that impose different data rates and latencies among compute nodes, we improve the random technique by distinguishing the "nearby" and "distant" neighbors. The technique always tries to steal resources randomly from the "nearby" partitions first, and will turn to the "distant" partitions if experiences several failures in a row. For example, in a Torus network, we can set an upper bound of number of hops between two controllers. If the hop count is less than the upper bound, the two partitions are considered "nearby"; otherwise, they are "distant". In a fat-tree network, "nearby" partitions could be the sibling controllers that have the same parent.

We configure each controller to manage 50 slurmds (50:1 configuration). However, SLURM++ can be configured to have any homogeneous partition size (e.g. 1, 50, 100, 1024), and heterogeneous partition sizes. We compare SLURM with SLURM++ under "sleep 0" jobs of different job sizes. Even though the "sleep 0" workloads are not typical HPC jobs that use MPI for synchronization and communication, they are simple enough to help quantify the overheads of resource allocation and job scheduling, as the first step.

(1) Small-Job Workload (job size is 1 node)

The first workload just includes one-node jobs (essentially MTC jobs), and each controller launches 50 jobs. Therefore, when the number of controller is n (number of compute demons is 50*n), the total number of jobs is 50*n. This workload is used to test the pure job launching speed in the best case from the performance's perspective.

Fig 10 shows the performance results. We see that the throughput of SLURM first increases to a saturation point, and then has a decreasing trend as the number of nodes scales up (51.6 jobs/sec at 250 nodes, down to 39 jobs/sec at 500 nodes). This is because the processing capacity of the centralized slurmctld is limited, and it takes longer time for the slurmctld to launch jobs as the job count and system scale increase. On the other hand, the throughput of SLURM++ increases almost linearly with respect to the scale, and this trend is likely to continue at larger scales. At 500-node scale, SLURM++ can launch jobs 2.34X faster than SLURM (91.5 jobs/sec vs. 39 jobs/sec). Given the throughput trends of SLURM++ and SLURM, we believe the speedup will only grow as the scale is increased.

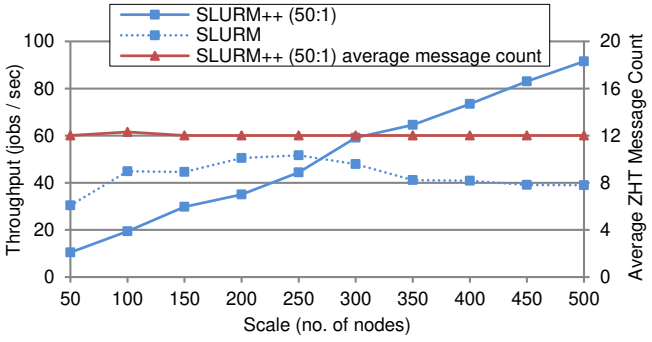


Fig 10: Small-Job: SLURM++ (50:1) vs. SLURM

The average ZHT message count per task remains almost constant with respect to the scale. This trend shows great scalability of SLURM++ for this workload. In prior work on evaluating ZHT [7], micro-benchmarks showed ZHT achieving more than 1M ops/sec at 1024K-node scale. At the largest scale of SLURM++, the average ZHT message count is 12 (about 6K messages for 500 jobs), along with the throughput of 91.5 jobs/sec, indicates ZHT message rate of 1098 ops/sec. ZHT is far from being a bottleneck for the workload and scale tested.

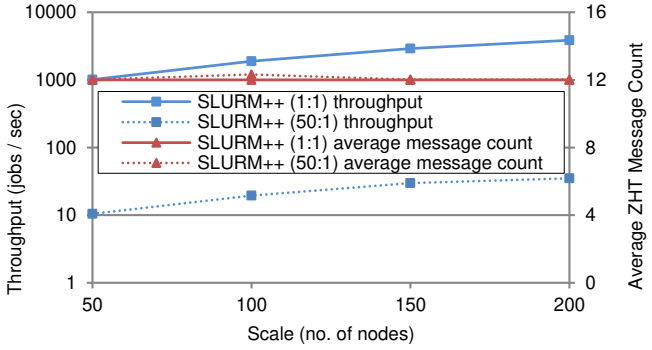


Fig 11: Small-Job: SLURM++ (1:1) vs. SLURM++ (50:1)

We also configure SLURM++ with 1:1 mapping of controllers to compute nodes to best support MTC workloads. We run experiments up to 200 nodes, with each controller launching a single one-node job. There are 200 controllers and 200 slurmds with 1:1 mapping. Fig 11 shows the performance results of SLURM++ with both 1:1 and 50:1 configurations. The throughputs increase linearly with respect to the scales, and the average message count keeps almost constant. For the 1:1 MTC configuration, based on

these trends, ideally, we can achieve 20K jobs/sec at 1K-node scale and will need to process only 12K ZHT messages. Besides, SLURM++ could be configured less aggressively with larger partition sizes, which would reduce the traffic loads to ZHT due to smaller number of controllers. Another fact is that the 1:1 configuration can achieve about 2 orders of magnitude higher throughput than the 50:1 one, even though the average message count doesn't change. This is because for 50:1 mapping, a controller needs to spend more time to fill the *value* for ZHT and to communicate with ZHT servers, as the length of *value* is longer, resulting in larger communication packages.

(2) Medium-Job Workload (job size is 1-50 nodes)

The second experiment tests how SLURM and SLURM++ behave under moderate job sizes. The workload is that each controller launches 50 jobs, and each job requires a random number of nodes ranging from 1 to 50.

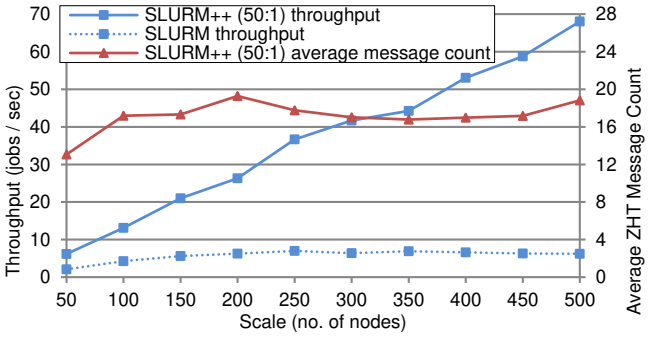


Fig 12: Medium-Job: SLURM++ (50:1) vs. SLURM

Fig 12 illustrates the performance results. We see that for SLURM, as the number of nodes scales up, the throughput increases a little bit (from 2.1 jobs/sec at 50 nodes to 7 jobs/sec at 250 nodes), and then keeps almost constant or with a slow decrease. For SLURM++, the throughput increases approximately linearly with respect to the scale (from 6.2 jobs/sec at 50 nodes to 68 jobs/sec at 500 nodes). SLURM++ can launch jobs faster than SLURM at any scale we evaluated, and the gap is getting larger at larger scales. At 500-node scale, SLURM++ achieves 11X (68 / 6.2) faster than SLURM; and the trends show that the speedup will increase at larger scales. We also see that the average ZHT message count first increases slightly (from 13 messages/job at 50 nodes to 19 messages/job at 200 nodes), and then experiences perturbations after that. The average ZHT message count will likely keep within a range (17-20), and might be increasing slightly at larger scales. This extra number of messages comes from the involved resource stealing operations.

(3) Big-Job Workload (job size is 25 – 75 nodes)

The third experiment tests the ability of both SLURM and SLURM++ of launching big jobs. In this case, each controller launches 20 jobs, where each job requires a random number of nodes ranging from 25 to 75.

The performance results are shown in Fig 13. SLURM shows a throughput increasing trend up to 500 nodes (from 1.2 jobs/sec at 100 nodes to 4.3 jobs/sec at 500 nodes), and the throughput is about to saturate after 400 nodes (from 3.8 jobs/sec at 400 nodes to 4.3 jobs/sec at 500 nodes). While the throughput of SLURM++ keeps increasing almost linearly up to 500 nodes. Like the mid-job case, SLURM++ can launch jobs faster than SLURM at any scale we evaluated, and the gap is getting larger as the scale increases. At 500-node scale, SLURM++ launches jobs about 4.5X (19.3 / 4.3) faster than SLURM. Again, we believe the speedup will become bigger at larger scale.

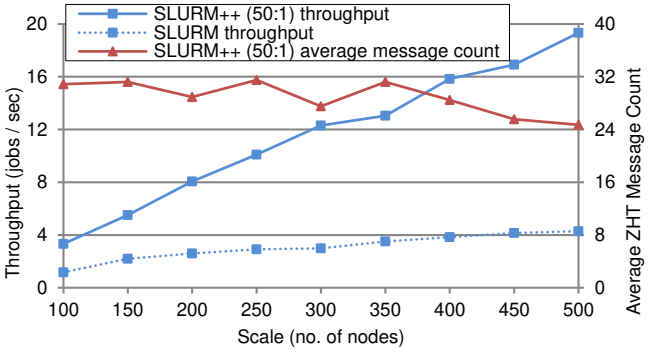


Fig 13: Large-Job: SLURM++ (50:1) vs. SLURM

In terms of the average ZHT message count, it shows a decreasing trend (from 30.1 messages/job at 50 nodes to 24.7 messages/job at 500 nodes) with respect to the scale. This is because when adding more partitions, each job that needs to steal resource will have higher chance to get resource as there are more options. This gives us intuition about how promising the resource stealing algorithm will solve the resource contention problems of distributed resource manager towards exascale computing.

We observe that not only does SLURM++ outperform SLURM in nearly all cases, but the performance slowdown due to increasingly larger jobs at large scale is better for SLURM++; this highlights the better scalability of SLURM++. Another fact is that as the scale increases, the throughput speedup is also increasing for all of the three workloads. This indicates that at larger scales, SLURM++ would outperform SLURM even more. Also, we can conclude that SLURM++ has great scalability for medium-size jobs, and there are improvements for large-size jobs. We have high hopes that distributed HPC scheduling can revolutionize batch-scheduling at extreme scales in the support of a wider variety of applications.

5.4.3 MATRIX Distributed Task Scheduler

MATRIX is a fully distributed task scheduler for fine-grained MTC workloads that include loosely coupled small (e.g. per-core) tasks with shorter durations (e.g. sub-second) and large volumes of data with dependencies. MATRIX uses work stealing [50] to achieve load balancing, and ZHT to keep the task metadata (data dependencies, data localities) in the support of monitoring task execution progress and data-aware scheduling.

Each scheduler in MATRIX maintains four queues (i.e. task wait queue, dedicated local task ready queue, shared work stealing task ready queue and task complete queue), and tasks are moved from one queue to another when state changes during execution. The task metadata is stored in ZHT, and modified when a task's state has changed. Tasks in the dedicated ready queue are scheduled and executed locally, while tasks in the shared work stealing queue could be migrated among schedulers for

balancing loads. A ready task will be put in either queue based on the size and location of the demanded data.

We evaluate MATRIX with the all-pairs application in biometrics [51]. All-Pairs application describes the behavior of a new function on two sets. For example, finding the covariance of two gene code sequences. In this workload, all the tasks are independent, and each task runs for 100 ms to compare two 12MB files with one from each set. We run strong-scaling experiments up to 200 cores using a 500*500 workload size with 250K tasks in total.

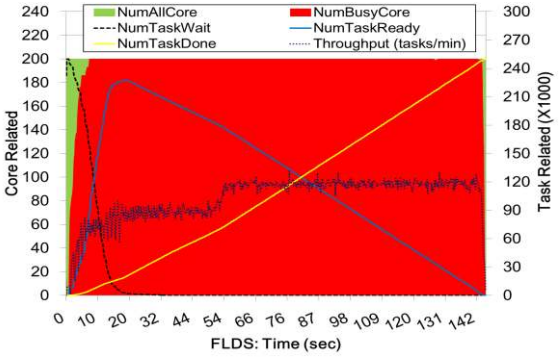


Fig 14: MATRIX ran all-pairs applications (200 cores)

The results are shown in Fig 14 at 200-core scale. We see that MATRIX can achieve almost 120K task/min (almost 2K task/sec) at the stable stage (after 54 sec) when the loads are perfectly balanced. This indicates a nearly 100% efficiency, as the throughput upper bound is 2K for 100ms tasks with 200 cores (200 * (1 / 0.01)). This also means the task launch time for fine-grained tasks is negligible comparing to the 100ms running time. The overall system utilization is about 95% (red area/green area), the 5% performance loss (100/0.95–100=5.3 ms extra time per task) is due to the short ramp up period needed to achieve load balancing and the very short task launching overhead. This great result attributes to the load balanced work stealing technique and the use of the ZHT KVS to store the task metadata in a distributed and scalable way.

5.4.4 Fault Tolerance of SLURM++ and MATRIX

Both SLURM++ and MATRIX have distributed architectures that apply multiple servers to participate in resource allocations and job scheduling. The clients can submit workloads to arbitrary server. The server failures would have trivial side effects on the functionalities offered to the clients, because the clients can easily re-submit workloads to another server in the case of server failures.

In terms of preserving and recovering the system state under server failures, since the servers of both SLURM++ and MATRIX are stateless (all the data is stored in ZHT), they can tolerate failures with a minimum efforts. We expect the ZHT to take over the responsibilities of dealing with replications, failure and recovery and consistency of the stored data. Up to date, ZHT has implemented different failure, recovery and consistency mechanisms.

Both systems demonstrate that KVS is a viable building block. Relying on KVS for distributed state management can not only ease the development of a general system software, but can improve the scalability, efficiency and fault tolerance of a system software significantly.

6 Related Work

Work that is related to the simulation of system software includes an investigation of peer-to-peer networks [52], telephony simulations [53], simulations of load monitoring [54], and simulation of consistency [55]. However, none of the investigations focused on HPC, or combine replication, failures and consistency. This survey [56] investigated 6 distributed hash tables and categorized them in a taxonomy of algorithms. The work focused on the overlay networks. In [57], p2p file sharing services were traced and used to build a parameterized model. Another taxonomy was developed for grid computing workflows [58]. The taxonomy was used to categorize existing grid workflow managers to find their common features and weaknesses. But none of these work targeted HPC workloads and system software, and none of them use the taxonomy to drive features in a simulation.

Examples of the types of system software of interest in HPC are listed in Table 1. It includes resource manager, SLURM [29], which is scalable for clusters. It has a centralized manager that monitors resources and assigns work to compute daemons; I/O forwarding system, IOFSL [30], which is a scalable, unified I/O forwarding framework for HPC systems; interconnect fabric managers, OpenSM [59]. OpenSM is an InfiniBand subnet manager; and data aggregation system, such as MRNet, which is a software overlay network that provides multicast and reduction communications for parallel and distributed tools. These are the types of system software that will be targeted for design explorations with our simulator.

Distributed key-value storage system is a building block for system software. Dynamo [33] is a highly available and scalable KVS of Amazon. Data is partitioned, distributed and replicated using consistent hashing, and eventual consistency is facilitated by object versioning. Voldemort is an open-source implementation of Dynamo developed by LinkedIn. Cassandra [34] is a distributed KVS developed by Facebook for Inbox Search. ZHT [48] is a zero-hop distributed hash table for managing the metadata of future exascale distributed system software. Our simulator is flexible enough to be configured to represent each of these key-value storage systems.

7 Conclusions and Future Work

The goal of this work was to propose a general system software taxonomy for exascale HPC system software, and to ascertain that a specific HPC system software should be implemented at certain scales with certain levels of replication and consistency as distributed systems. We devised a *system software taxonomy*. Four classes of system architectures were studied through a *key-value store simulator*. We conducted extreme-scale experiments to quantify the overheads of different recovery, replication and consistency models for these architectures. We also showed how KVS could be used as a building block for general system software. The motivation was that a centralized server architecture doesn't scale and is a single point of failure. Distributed system architectures are necessary to expose the extreme parallelism, to hide latency, to maximize locality, and to build scalable and reliable system software at exascale.

The conclusions of this work are: (1) KVS is a viable building block; (2) when there are a huge amount of client requests, d_{fc} scales well under moderate failure frequency, with different replication and consistency models, while d_{chord} scales moderately with less expensive overhead; (3) when the communication is dominated by server messages (due to failure/recovery, replication and consistency), d_{chord} will have an advantage; (4) different consistency models have different application domains. Strong consistency is more suitable for running read-intensive applications, while eventual consistency is preferable for applications that require high availability (shown in Fig 15).

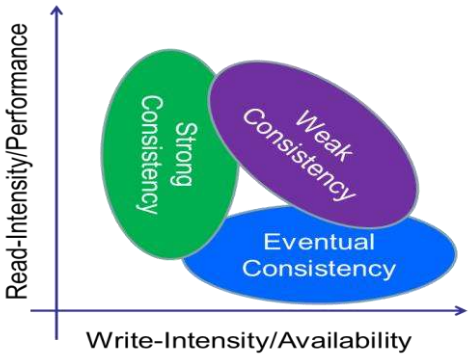


Fig 15: Guide to choose different consistency models

The tools that are developed allow insights to be made in scalable system software design through simulations. We show how these insights can be applied in the design of real system software, such as SLURM++ and MATRIX, by leveraging KVS as a building block to reduce complexity in developing modern scalable system software.

Future work includes improving the comprehensiveness of the taxonomy. For example, the network model will consider the latencies and data rates of links of different kinds of interconnections (e.g. Ethernet and InfiniBand); it should also consider more complicated topologies, such as Dragonfly, Fat tree, and multi-dimension Torus networks. We also plan to evolve the simulator to cover more of the taxonomy. Furthermore, we will use the simulator to model other system software and validate these at small scale, and then simulate at much larger scales. This work will guide the development of a general building block library that can be used to compose large scale distributed resilient system software. Besides the distributed resource manager (SLURM++) and task scheduler (MATRIX), other system software implementations will be developed to support c_{single}, c_{tree}, and d_{chord} with various properties from the taxonomy.

ACKNOWLEDGMENT

This work was supported by the U.S. Department of Energy under contract DE-FC02-06ER25750, and in part by the National Science Foundation under award CNS-1042543 (PRObE). This work was also in part supported by the National Science Foundation grant NSF-1054974. This research also used resources of the ALCF at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DEAC02-06CH11357.

REFERENCES

[1] Department of Energy. Architectures and Technology for Extreme Scale Computing. San Diego, CA: Department of Energy, 2009.

[2] P. Kogge, K. Bergman, et al. ExaScale computing study: Technology challenges in achieving exascale systems, Sept. 28, 2008.

[3] M. A. Heroux. Software challenges for extreme scale computing: Going from petascale to exascale systems. Int. J. High Perform. Comput. 2009.

[4] Department of Energy. Top Ten Exascale Research Challenges. DOE ASCAC Subcommittee Report. February 10, 2014.

[5] Department of Energy. Exascale Operating System and Runtime Software Report. DOE Office of Science, Office of ASCR, November 1, 2012.

[6] R. Wisniewski. HPC System Software Vision for Exascale Computing and Beyond. Salishan conference. April 23, 2014.

[7] X. Besseron and T. Gautier. "Impact of Over-Decomposition on Coordinated Checkpoint/Rollback Protocol", Euro-Par Parallel Processing Workshops, Lecture Notes in Computer Science Volume 7156, 2012.

[8] R. Rabenseifner, G. Hager, G. Jost. "Hybrid MPI and OpenMP Parallel Programming", Tutorial tut123 at SC13, November 17, 2013.

[9] ZeptoOS project. http://www.zeptoos.org/. MCS of ANL, 2015.

[10] Kitten Lightweight Kernel. https://software.sandia.gov/trac/kitten. Sandia National Laboratory, 2015.

[11] L. Kale, A. Bhatele. "Parallel Science and Engineering Applications: The Charm++ Approach". Taylor & Francis Group, CRC Press. 2013.

[12] M. Bauer, S. Treichler, et al. Legion: expressing locality and independence with logical regions. ACM/IEEE, SC '12.

[13] H. Kaiser, T. Heller, et al. HPX – A Task Based Programming Model in a Global Address Space. International Conference on PGAS 2014.

[14] Antal Buss, Harshvardhan, et al. STAPL: standard template adaptive parallel library. In Proceedings of the 3rd SYSTOR conference, 2010.

[15] J. Wozniak, T. Armstrong, et al. Foster Swift/T: Large-scale application composition via distributed-memory data flow processing. In Proc CCGrid 2013.

[16] H. Kaiser, M. Brodowicz, et al. "ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications," In Parallel Processing Workshops, 2009.

[17] N. Thomas, S. Saunders, et al. ARMI: A high level communication library for STAPL. Parallel Processing Letters, 16(02):261–280, 2006.

[18] J. Lidman, D. Quinlan, et al. "Rose: fttransform-a source-to-source translation framework for exascale fault-tolerance research," in 42nd Dependable systems and networks workshops (dsn-w), 2012.

[19] M. Kulkarni, A. Prakash, M. Parks. "Semantics-rich Libraries for Effective Exascale Computation or SLEEC". https://xstackwiki.modelado.org/SLEEC. 2015.

[20] H. Weatherspoon and J. D. Kubiatowicz. "Erasure coding vs. replication: A quantitiative comparison", Proc. IPTPS, 2002.

[21] W. Vogels. 2009. Eventually consistent. Commun. ACM 52, 1, 2009.

[22] K. Wang, X. Zhou, et al. "Next Generation Job Management Systems for Extreme Scale Ensemble Computing", ACM HPDC 2014.

[23] K. Wang, A. Rajendran, I. Raicu. "MATRIX: MAny-Task computing execution fabRIc at eXascale," tech report, IIT, 2013.

[24] K. Wang, X. Zhou, et al. "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling", IEEE BigData 2014.

[25] K. Wang, A. Rajendran, et al. "Paving the Road to Exascale with Many-Task Computing", Doctoral Showcase, SC 2012.

[26] K. Wang, A. Kulkarni, et al. "Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services", IEEE/ACM Supercomputing/SC 2013.

[27] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. ACM Comput. Surv. 15, 4, 1983, 287-317.

[28] I. Stoica, et al. Chord: A scalable peer-to-peer lookup service for

internet applications. Sigcomm Comput. Commun. Rev., 2001.
[29] M. Jette, A. Yoo, et al. SLURM: Simple Linux utility for resource management. International Workshop on JSSPP, 2003.
[30] N. Ali, P. Carns, et al. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In CLUSTER, 2009.
[31] A. Vishnu, A. Mamidala, et al. Performance Modeling of Subnet Management on Fat Tree InfiniBand Networks using OpenSM. In IPDPS'05 - Workshop 18 - Volume 19, 2005.
[32] P. Roth, D. Arnold, et al. MRNet: A software-based multicast/reduction network for scalable tools. ACM/IEEE SC'03.
[33] G. DeCandia, et al. Dynamo: Amazon's highly available key-value store. In Proceedings of ACM SOSP, 2007.
[34] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 2010.
[35] A. Feinberg. Project Voldemort: Reliable Distributed Storage. ICDE, 2011.
[36] M. A. Heroux. Toward Resilient Algorithms and Applications, April 2013. Available from http://www.sandia.gov/~maherou/docs/HerouxTowardResilientAlgsAndApps.pdf
[37] I. Raicu, I. Foster, et al. Middleware support for many-task computing. Cluster Computing, 13(3):291–314, September 2010. ISSN: 1386-7857.
[38] I. Raicu, I. T. Foster, et al. Many-task computing for grids and supercomputers. In MTAGS workshop, 2008.
[39] Ioan Raicu. Many-task computing: Bridging the gap between high-throughput computing and high-performance computing. Proquest, Umi Dissertation Publishing, 2009.
[40] I. Raicu, et al. Towards data intensive many-task computing. In Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2009.
[41] D. Zhao, Z. Zhang, et al. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems", IEEE BigData 2014.
[42] D. Karger, E. Lehman, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of STOC '97.
[43] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In Proceedings international conference on Simutools, 2008.
[44] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In IEEE Global Internet Symposium, 2007.
[45] T. Vignaux and K. Muller. Simpy:documentation, May 2010. Available from http: //simpy.sourceforge.net/SimPyDocs/index.html
[46] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), 2009.
[47] K. Wang, K. Brandstatter, et al. SimMatrix: Simulator for many-task computing execution fabric at exascale. In Proceeding of HPC'13.
[48] T. Li, X. Zhou, et al. ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table. IPDPS, 2013.
[49] M. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1):124–149, 1991.
[50] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 1999.
[51] I. Raicu, I. Foster, et al. "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems", ACM HPDC09, 2009.
[52] Ti. Tuan, A. Dinh, et al. Evaluating Large Scale Distributed Simulation of P2P Networks. In Proceedings of the 2008 12th IEEE/ACM International Symposium on DS-RT, 2008.
[53] I. Diane, I. Niang, et al. A Hierarchical DHT for Fault Tolerant Management in P2P-SIP Networks. In Proceedings of the 2010 IEEE 16th International Conference on ICPADS, 2010.
[54] B. Ghit, F. Pop, et al. Epidemic-Style Global Load Monitoring in Large-Scale Overlay Networks. In Proceedings of the 2010 International Conference on 3PGCIC, 2010.
[55] M. Rahman, W. Golab, et al. Toward a Principled Framework for Benchmarking Consistency. In Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability (HotDep'12). 2012.
[56] E. Lua, J. Crowcroft, et al. A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials, 7(2):72–93, 2005.
[57] K. Gummadi, et al. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In Proceedings of the nineteenth ACM symposium on SOSP, 2003.
[58] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. Journal of Grid Computing, 3(3-4):171–200, 2005.
[59] A. Vishnu et al. Performance modeling of subnet management on fat tree infiniband networks using opensm. Workshop 18, IPDPS, 2005.

Ke Wang is a Ph.D. candidate of the Department of Computer Science at Illinois Institute of Technology (IIT). He is a member of the Data-Intensive Distributed Systems Laboratory at IIT, and working with Dr. Ioan Raicu. His research work and interests are delivering distributed job management systems towards extreme-scale computing. Wang received his degree of B.S. in Software Engineering from Huazhong University of Science and Technology, Wuhan, China in 2010. He is a student member of IEEE and ACM.

Abhishek Kulkarni is a PhD candidate at the School of Informatics and Computing at Indiana University. He is a member of the Center for Research in Extreme Scale Technologies (CREST) working with Dr. Andrew Lumsdaine. His research interests include execution models and runtime systems for HPC, and performance modeling and simulation of parallel programs. He received his MS in Computer Science from Indiana University in 2010.

Michael Lang is the team leader of the Ultrascale Systems Research at Los Alamos National Laboratory (LANL). His research interests include interconnects for large-scale systems, performance of large-scale systems, operating system and runtime issues for exascale and HPC. Lang was formerly a member of LANL's Performance and Architecture team, involved in performance analysis of large-scale systems for DOE. He received a B.S. in Computer Engineering and an M.S. in Electrical Engineering in 1988 and 1993 respectively, both from the University of New Mexico.

Dr. Dorian Arnold is an assistant professor in the Department of Computer Science at the University of New Mexico. He is co-director of the Scalable Systems Laboratory at UNM where he directs research in the broad areas of high-performance computing and large scale distributed systems, with focuses on scalable middleware, runtime data analysis, fault-tolerance and HPC tools. Arnold collaborates with researchers from several national laboratories and universities, and his research projects have been selected as Top 100 R&D technologies in 1999 and 2011. He holds a Ph.D. in Computer Science from the University of Wisconsin, an M.S. in Computer Science from the University of Tennessee and a B.S. in Mathematics and Computer Science from Regis University.

Dr. Ioan Raicu is an assistant professor in the Department of Computer Science at IIT, as well as a guest research faculty in the Math and Computer Science Division at Argonne National Laboratory. He is also the founder (2011) and director of the Data-Intensive Distributed Systems Laboratory at IIT. He obtained his Ph.D. in Computer Science from University of Chicago under the guidance of Dr. Ian Foster. He is particularly interested in many-task computing, data intensive computing, Cloud computing, and many-core computing. He is a member of the IEEE and ACM.