

A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems

Matthew G. F. Dosanjh and Patrick G. Bridges
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131
{mdosanjh,bridges}@cs.unm.edu

Suzanne M. Kelly and James H. Laros III
Scalable System Software Department
Sandia National Laboratories
Albuquerque, NM 87185-1319
{smkelly,jhlaros}@sandia.gov

Abstract—Historically, scientific computing applications have been statically linked before running on massively parallel High Performance Computing (HPC) platforms. In recent years, demand for supporting dynamically linked applications at large scale has increased. When programs running at large scale dynamically load shared objects, they often request the same file from shared storage. These independent requests tax the shared storage and the network, causing a significant delay in computation time. In this paper, we propose to leverage a proven file sharing technique, BitTorrent, abstracted by an on-node FUSE interface to create a system-level distribution method for these files. We detail our proposed methodology, related work, and our current progress.

I. INTRODUCTION

High Performance Computing (HPC) applications are increasingly using demand-loaded dynamic shared libraries (DSLs) at very large scales. This is a relatively new requirement for HPC system software; in the past, HPC programs were generally statically compiled, removing the need to load DSLs from remote filesystems over the network. DSLs are attractive in HPC systems, however, because they can be used to support high-level programming and scripting languages, as well as for functionality such as loading solver modules on-demand based on application need. They also can substantially reduce the memory footprint of scientific applications, which are reaching 1GB in size when statically linked.

Supporting demand-loading of shared libraries in large-scale systems is challenging. Standard file system techniques that work well on a single node and small scale platforms work poorly at very large scale; 50,000 simultaneous requests for the same file is very difficult to handle quickly even on the most scalable HPC file systems. Specialized techniques for supporting dynamic libraries in HPC systems also have scaling problems,

with recent work documenting sub-optimal initial load times. Two reports, one using the Kull application, and the other its benchmark standin, Pynamic [1], report load times on the order of twenty minutes for 16,000 processes. Data from one report came from an IBM Blue Gene/P system [2], the other from a Cray XE system [3].

In this paper, we propose to use modern peer-to-peer data distribution techniques based on the BitTorrent protocol [4] to support large-scale loading of DSLs. In this approach, users specify the set of libraries they would like to have available at runtime, these files are assembled into a BitTorrent archive, and a torrent description and relevant meta-data is distributed to nodes at job launch. The BitTorrent protocol is then used to scalably move files to compute nodes at runtime based on application demand for the files.

The remainder of this paper is organized as follows. Section II describes the problem in more detail, and Section III then describes our approach and the architecture of our proposed system. Section IV details our current progress towards evaluating our approach, and Section V describes optimizations to the approach that we plan to explore. Finally, Section VI discusses previous work done in the area and Section VII concludes.

II. BACKGROUND

While developed to increase resource efficiency for multi-user timeshared desktop and enterprise level platforms, the demand for shared library support on clusters and large scale High Performance Computing (HPC) platforms has greatly increased. Many of the design choices made, while appropriate for the target design space, are obstacles to scalability, especially on large clusters and HPC platforms. Other features, like multiple executables sharing a single dynamically loaded object, are of no benefit in a space-shared environment.

When an object is statically linked, all references are resolved at compile time and all executable code is contained in the resulting binary. This binary can be efficiently distributed to tens of thousands of nodes in several minutes, which is attractive for HPC systems. While distributing a single object at launch has its advantages, the increasing use of large solver libraries often results in huge executables. Since code paths are decided based on the specific problem or input, large portions of unused code must be statically built into executables which can bloat executables with unnecessary code.

Dynamically linked binaries defer most of the linking process until run-time. When a binary is executed the dynamic linker attempts to resolve all unresolved references. This process causes a large number of file stat operations. This large volume of file operations is the first key challenge of this problem. While on a single time-shared system these operations can be efficiently accomplished, on a large HPC platform they scale with the number of concurrent processes. This can be tens of thousands of simultaneous file operations for the same set of shared libraries on a single shared filesystem to resolve references. These operations are latency-sensitive and handling them quickly is essential. Even small delays can cause timeouts which result in duplicate requests, further increasing load on the file server. Once all references are resolved, the program can begin execution.

Once the program is executing, if any of the shared object code resolved during the initial run-time linking process is required, the dynamic loader is employed to service the request. Typically, this is done using the OS file-system interface. Requested objects which were memory mapped during the linking phase are demand paged. Again, this process is efficient for single time-shared systems. On HPC systems, bulk synchronous executions make requests roughly at the same time. Because of this, the scenario of tens of thousands of nodes making simultaneous requests to a shared object becomes plausible. All of the processes will request this page (or pages) of the shared object at roughly the same time from the single shared library on the same shared file-system. The bandwidth and meta-data coordination of this data distribution poses the second key challenge.

Possible solutions to these key challenges are unfortunately quite disparate. Parallel file-systems are typically optimized for delivering large amounts of bandwidth. While this seems encouraging on the surface, parallel file-system performance for shared files (N to 1) is typically much worse than can be achieved in an N to

N organization. File operations like directory searches, file stats and opens often expose bottlenecks in parallel file-system meta-data services.

As a result, modern HPC systems, for example the Cray XT/XE/XK line of supercomputers, generally use a hierarchy of file caches for system shared libraries. The recent Alliance for Computing at Extreme Scale (ACES)¹ capability platform, Cielo, uses the same strategy for user-provided shared libraries. This approach is more efficient than any other available file system on Cielo [5]. However, even when using this more efficient hierarchy of caches, the application runtime can still increase substantially when compared with a statically linked binary [3].

III. APPROACH AND ARCHITECTURE

There are three main features our system must provide based on how demand-shared libraries are used in HPC systems:

- 1) Data distribution to nodes that can handle moving large shared libraries to tens or hundreds of thousands of nodes without bottlenecking a single or small number of file system nodes
- 2) Meta-data management that can handle repeated directory queries for searching the file system for appropriate shared libraries (e.g. LD_LIBRARY_PATH searches)
- 3) Straightforward integration with the compute-node operating systems so that the resulting system can be deployed to and maintained for production systems with minimal extra effort

In the remainder of this section, we describe the overall architecture of the system on which we are working and our approach to addressing the challenges above.

A. Overall Architecture

As mentioned above, our general approach to supporting demand-shared libraries for large-scale supercomputers is to leverage peer-to-peer data distribution techniques, in particular BitTorrent. Techniques like BitTorrent are particularly appropriate for supporting shared library-oriented file systems because they are built for read-only data.

In the general approach, shown in Figure 1 user actions take place when specifying a job to run, when the job is launched, and when requests are made at runtime for the requested files. When constructing a job

¹ACES is a collaboration between Sandia National Laboratories and Los Alamos Laboratory to design and field capability class platforms for the DOE NNSA Tri-lab community

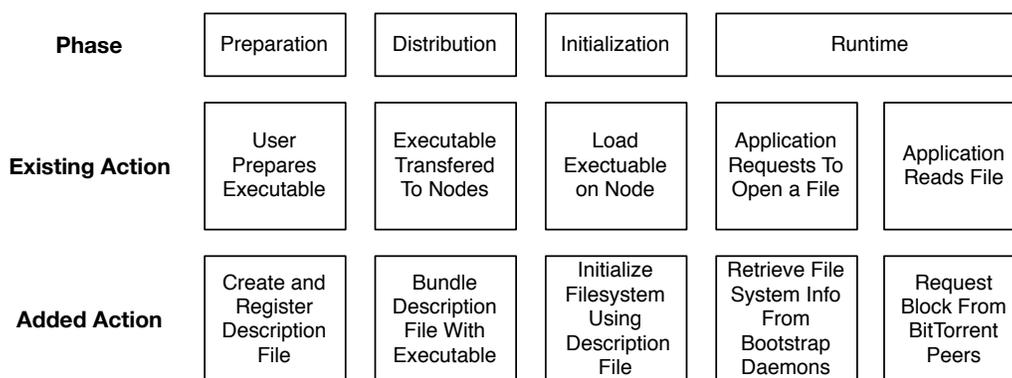


Figure 1. Steps in System Operation

to submit, users will declare a set of files, for example a directory of shared object files, to make available to the application. Our system will create a small *description file* that contains information needed about the files to make available.

When a job is launched, the description file will be distributed to each compute node as part of the launch process; this file will specify what local directory on the compute node the files should be mounted in, as well as all of the information necessary to handle file lookups at runtime, to begin the transfer of data to compute nodes when data is requested, and to serve, in a peer-to-peer fashion, requests from other compute nodes. In addition, *bootstrap daemons* on system service or I/O node will also process the description file so that they can handle requests from compute nodes to bootstrap data movement onto the compute nodes for peer-to-peer distribution.

As the job runs, metadata and data requests (e.g. resulting from a call to `dlopen` library function) will be handled by communicating with other compute node and, when necessary, with bootstrap daemons on service nodes. We describe the details of our approach to handling such requests in the remainder of this section.

B. Data Movement

We propose to use a peer-to-peer data distribution protocol based on the BitTorrent protocol to handle the growth of data distribution requirements with increasing system scale. BitTorrent allows nodes that have previously requested data to provide data to their peers. This will allow compute nodes that have previously accessed a shared library to provide data to other compute nodes. This will remove the dependence on shared storage and allow the distribution to scale.

We have identified two different ways to transfer the data between the nodes. The first is at a file granularity. BitTorrent allows for priorities to be set for individual files. When a request to open a file is given, we download the entire file to RAMDISK. The other option is to use a block granularity. BitTorrent separates all of the data into arbitrarily sized blocks, these can be requested from the other nodes. Using this, our system requests a block when it is read from. To do this, the read must be mapped from disk blocks to BitTorrent blocks to determine which data to download.

In the first approach, the cost is encountered when a file is opened. However, reading the file becomes less expensive because the data is already stored locally. In the second approach, the cost is encountered when a read accesses new data. By only downloading the requested block, we can avoid downloading unnecessary data. We plan to implement and evaluate both options.

The use of a simple peer-to-peer protocol will allow this protocol to scale up to extremely large node counts. Some features of BitTorrent, particularly those related to free-loader prevention, will be disabled because they are not needed in the cooperative context of a single HPC application. Open source BitTorrent libraries will be used to facilitate implementation of this portion of the system.

C. Meta-data Management

We identified two options to handle metadata. First is to create a file that contains relevant information and to have compute nodes download this from the BitTorrent swarm. The other option is to use the information stored in the description file to generate metadata on the node. This second option may also allow us to cache more metadata in the description file, allowing us to add metadata that is not already there. Since a BitTorrent description file stores some information, such as directory

structure and filesize, we will identify what metadata can be reconstructed from the description file and determine what metadata we need to transfer to the node. We must ensure that security-sensitive (e.g. ownership and permission) metadata information is obtained from a trusted source.

D. Bootstrapping

Our system must complete a bootstrapping process before a target application can access it. The bootstrapping process requires two bootstrapping daemons to initialize our system:

- 1) A BitTorrent tracker must be running on a I/O node. The tracker registers torrents, maintains a list of peers and serves as an entry point into the BitTorrent swarm. The tracker must be running before the description file can be registered.
- 2) Initial seeders are responsible for fetching the data and serving the initial requests from the compute nodes. These run on an I/O node. They must fetch and store the data before initializing the BitTorrent library. When BitTorrent is initialized, it will recognize and verify the files. Upon completion it will register with the tracker as a seeder.

E. OS Integration

Because we want to make our system transparent to the application programmer, integration with the OS is necessary. To integrate into the operating system we are using a filesystem framework called FUSE (Filesystem in Userspace). FUSE provides an interface by which developers can override filesystem functions to provide different types of functionality. By implementing a FUSE module, we can implement our system, remaining transparent to the application programmers.

To illustrate our FUSE integration, we can follow the data path given a file granularity. Initially the application makes a request for a file, such as in a `dlopen` call. This request is routed through the kernel to the FUSE module. The FUSE module, upon receiving a request to open a file, will request the file from the BitTorrent library. The BitTorrent library will change the priority of these files from undesired to normal or high. The BitTorrent library will then start actively looking for the file from the BitTorrent swarm, saving the file into a RAMDISK. After the file is downloaded, FUSE will provide an interface for the application to access the file on the RAMDISK.

IV. CURRENT STATUS

We have begun work on constructing the system described in the previous section. Our initial goals are to implement a proof of concept and use it to collect initial performance data to demonstrate the viability of the approach and drive later optimization decisions. In the remainder of this section, we describe our progress on implementing the described approach, the next steps in this developing a prototype version of this approach, and our plans for testing and evaluating the system.

A. Required FS Operations

As a first step, we examined the FUSE functions called when using a DSL, both implicitly by the system loader on program launch and explicitly through `dlopen`. We did this by running Pynamic [1], a DSL benchmarking tool, on a FUSE module that logged each request that it received [6]. In addition, we added the FUSE directory to the `LD_LIBRARY_PATH` to examine dynamic library-related file system calls on program launch.

On program launch, the system dynamic linker resolves shared libraries referenced by the program executable. To do this, it makes a `stat()` system call on each directory in the given path and, assuming the directory exists, on the name of each shared library for which it is searching in that directory. This results in a `get_attributes` request to FUSE. `dlopen()` similarly uses `stat()` to find if the file it is searching for exists as opposed to searching the containing directory for the appropriate name directly. Overall, this generates a large volume `stat()` system calls which we will have to focus on optimizing in our work.

After locating the appropriate library, the dynamic linker attaches it to a running program using system calls that result in FUSE `open`, `flush`, and `release` operations. These operations are requested once per dynamically linked library accessed. `get_attributes` is requested more often, by about 1.6 times. As the program uses functions in the loaded object, the `read` operation is invoked for each accessed page. The number of invocations to `read` and the rate at which it is called depends on the behavior of the program using the shared object.

B. Implementation Progress

We are now focusing on prototyping BitTorrent-based data movement in a FUSE-based file system implementation for compute nodes. The first challenge we have faced in this process has been to find a BitTorrent library that is appropriate for use in HPC systems. In particular, we need a library that supports modern

BitTorrent features (such as trackerless torrents based on distributed hash tables) for future optimization work, and is simple and open-source so that we can use it for production compute nodes with minimal footprint or dependencies. After evaluating a selection of BitTorrent libraries, we found that libTorrent [7] best fit our needs.

Our initial work on combining libTorrent with FUSE is using a simple local RAMDISK approach to transferring data that transfers and caches entire files locally. Specifically, when FUSE opens a particular shared library file, our current implementation will transfer the entire file into a local RAMDISK so that later read requests can be immediately satisfied locally. As mentioned in Section III-B, this approach makes application launch and `dlopen()` an expensive operation and increases the memory footprint of our prototype. On the other hand, this dramatically reduces the cost of later requests to read from the file as, for example, the application begins executing in the shared library.

In addition, our initial implementation will satisfy attribute requests (e.g. directory lookups) file stat requests by transferring i-node data for each file referenced by a directory when the directory itself is opened. This will allow us to handle both directory name search requests that read the directory itself and `stat()` operations on the files, with minimal network traffic. We will examine additional optimizations as necessary, for example embedding file attributes in the description file distributed at launch time as described later Section V,

C. Next Steps

After getting basic BitTorrent/FUSE data movement working, our next step will focus on constructing the infrastructure to allow this approach to be used on HPC systems. In particular, we will focus on setting up the bootstrap tracker and seeder daemons needed to start data transfer in the system, as well as the tools for constructing and distributing torrent files at application build and launch. Our initial work will focus on launching these bootstrap daemons on a per-application basis, probably using compute node rank 0 to host these processes. We expect that this will overburden rank 0 at large scales, as well as increase the noise that the application code running on rank 0 experiences, so we will later work on constructing system-wide tracker and seeder daemons that can be run on I/O nodes.

D. Testing Strategy

To evaluate our system, we will use both the Pynamic benchmark [1] to examine dynamic loading of large

numbers of shared libraries at runtime, and dynamically linked version of a variety of HPC benchmarks and applications, including the HPC Challenge [8] and NAS Parallel [9] benchmarks. We will begin running these applications on small development clusters to better understand the impact of the design choices in our approach on their runtime. After initial development, optimization, and performance tuning at small to moderate scales, we plan to also evaluate the viability of this approach on larger scale production HPC systems with thousands or tens of thousands of nodes.

V. FUTURE OPTIMIZATIONS

There are a number of optimization and performance tradeoffs we plan to study as part of this work. First, BitTorrent has recently moved to a trackerless system for managing the location of data in the system, where a distributed hash table is used to track this information. Removing the need for a tracker daemon that all processes need to contact to locate data to transfer is a potentially important optimization in our system. As a result, exploring the performance benefits of a trackerless BitTorrent implementation is an optimization we plan to examine as soon as possible.

In addition, we also plan to study the performance tradeoffs between block-by-block transfer of data versus whole-file caching of files. Block-by-block transfer of data reduces the time taken to initially opening a shared object, potentially reducing application startup time. It also reduces the memory footprint of the shared library file system dramatically. However, deferring transfer of data potentially increases variance in application execution time because shared library contents will be need to be transferred at a later time as these pages are touched. This potentially increases system noise, which can also dramatically impact application runtime [10]. As a result, we also plan to examining the tradeoffs of different caching strategies and granularities at varying system scales.

Finally, attribute lookups are another potential area that may need to be optimized in our system. As mentioned in Section III-C, we may embed metadata for the complete file system into the description file to reduce metadata access times if this becomes a bottleneck in the system. As the most aggressive shared library systems (e.g. the pynamic benchmark mentioned in Section IV-D only uses a few thousand files at most, the amount of metadata about these files that would need to be embedded in the description file is quite modest.

VI. RELATED WORK

Research related to this effort comes primarily from two areas; HPC, the target of this research, and distributed systems. Sandia National Laboratories conducted some earlier experiments specifically directed towards supporting DSLs on HPC platforms that utilized light-weight kernels and custom run-time systems. While this experiment was generally successful, it required extensive changes to the standard loader and run-time and was not portable to other commodity software stacks. This effort, and a more general survey of other potential approaches, was documented in a Sandia technical report [11]. One of the alternative approaches posited in [11], again for use in a custom environment, was a proxy file-system approach.

To address scalable IO, the IO forwarding approach is used on Blue Gene P (Blue Gene L supported only statically compiled binaries) [12]. In [12], the authors focus on IO performance and bottlenecks and do not specifically address DSLs. Cray Inc. has also implemented DSL support using their file-system proxy, Data Virtualization Service (DVS) [13]. Coverage of Cray's approach and an optimization that extended the scalability of this approach (a DVS optimization) can be found in [5]. Other researchers have investigated optimizing IO on HPC platforms using proxy methods and data compression ([14], [15], and [16]). These techniques can improve the scalability of using DSLs but have not focused on this issue.

Magic Ermine [17], a tool developed to aid in binary portability, was also investigated in [11] as a possible hybrid method of combining both the executable and the required DSLs into a single package which could then be efficiently distributed on the target HPC platform. While interesting, this approach would generate very large combined executables along with other limitations outlined in [11].

In the distributed systems area there are a number of related research efforts. Chord [18], CAN [19], Pastry [20] and Tapestry [21]. These original four peer-to-peer Distributed Hash Table proposals were all introduced in 2001. All, primarily academic and research focused, share the same fundamental idea but have significantly different approaches. The original protocol for Bittorrent was also designed in 2001 but specifically targeted peer-to-peer file sharing over the Internet. We hope to benefit from some of the inherent characteristics of Bittorrent while ignoring some of the less applicable features.

Current efforts in file system R&D is important to monitor when addressing the problem of scalable access to dynamic shared libraries. File system choices for storing these shared libraries are limited for HPC systems. Current high performance file system software, such as Lustre, GPFS, PVFS, or Panasas, use parallel techniques to achieve good performance in aggregate. These packages are tuned for high volume, bursty I/O, such as writing checkpoint or reading restart files. The serial I/O to a single file on these parallel systems is not much better than a single disk file system. In fact, most shared libraries are stored on cluster-local disk or on network attached storage (NAS) and accessed via NFS. As mentioned previously, I/O forwarding techniques from NFS servers have been developed with some, but insufficient success. Newer efforts, targeting the Exascale timeframe, may alleviate some of the access issues. For example the Sirocco [22] effort is a peer to peer file system that may provide advantages similar to what we are proposing using Bittorrent.

VII. CONCLUSIONS

In this paper, we described a new peer-to-peer approach to supporting dynamic shared libraries in HPC systems that we are pursuing. We believe this approach has substantial potential for improving the support for applications that rely on dynamic linking and loading in HPC systems, and such application are increasingly important in HPC systems. We have begun implementing this approach using existing open source tools as the basis for our work, and are currently working towards an initial experimental evaluation of the potential of this approach.

ACKNOWLEDGEMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000.

This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program.

REFERENCES

- [1] G. L. Lee, D. H. Ahn, B. R. de Supinski, J. Gyllenhaal, and P. Miller, "Pynamic: the python dynamic benchmark," in *Proceedings of the IEEE 10th International Symposium on Workload Characterization*, Sep. 2007, pp. 101–106.

- [2] W. S. Futral, J. Gyllenhaal, and R. Hedges, "Level-2 Milestone 3504: Scalable Applications Preparations and Outreach for the Sequoia ID (Dawn)," Lawrence Livermore National Laboratory, Technical Report LLNL-TR-440586, 2010.
- [3] B. Barrett, R. Barrett, J. Brandt, R. Brightwell, M. Curry, N. Fabian, K. Ferreira, A. Gentile, S. Hemmert, S. Kelly, R. Klundt, J. H. Laros III, V. Leung, M. Levenhagen, G. Lofstead, K. Moreland, R. Oldfield, K. Pedretti, A. Rodrigues, D. Thompson, T. Tucker, L. Ward, J. V. Dyke, C. Vaughan, and K. Wheeler, "Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale," Sandia National Laboratories, Technical Report SAND2012-1750, March 2012.
- [4] B. Cohen, "The BitTorrent Protocol Specification," http://www.bittorrent.org/beps/bep_0003.html. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html
- [5] S. M. Kelly, R. Klundt, and J. H. Laros III, "Shared Libraries on a Capability Class Computer," in *Cray User Group Annual Technical Conference*, May 2011.
- [6] J. Joseph J. Pfeiffer, "Writing a FUSE Filesystem: a Tutorial." [Online]. Available: <http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/>
- [7] J. Sundell, "The libTorrent and rTorrent Project." [Online]. Available: <http://libtorrent.rakshasa.no/>
- [8] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) Benchmark Suite," in *SC06 Conference Tutorial*, 2006.
- [9] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NASA Ames Research Center, Moffett Field, CA, Tech. Rep. NAS-95-020, 1995.
- [10] K. B. Ferreira, R. Brightwell, and P. G. Bridges, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.
- [11] J. H. Laros III, S. M. Kelly, M. J. Levenhagen, and K. T. Pedretti, "Investigating Methods of Supporting Dynamically Linked Executables on High Performance Computing Platforms," Sandia National Laboratories, Technical Report SAND2009-5275073, 2009.
- [12] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii, "Accelerating I/O Forwarding in IBM Blue Gene/P Systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, Nov. 2010.
- [13] S. Sugiyama and D. Wallace, "Cray DVS: Data Virtualization Service," in *Cray User Group Annual Technical Conference*, May 2008.
- [14] B. Welton, D. Kimpe, J. Cope, C. Patrick, K. Iskra, and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," in *International Conference on Cluster Computing*. IEEE, Sept. 2011.
- [15] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization Techniques at the I/O Forwarding Layer," in *International Conference on Cluster Computing*. IEEE, Sept. 2010.
- [16] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *International Conference on Cluster Computing*. IEEE, Sept. 2009.
- [17] "Magic Ermine." [Online]. Available: <http://www.magicermine.com/erk/>
- [18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Special Interest Group on Data Communication (SIGCOMM)*, August 2001.
- [19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Special Interest Group on Data Communication (SIGCOMM)*, August 2001.
- [20] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.
- [21] B. Y. Zhao, K. J. D., and A. D. Joseph, "Tapestry: a fault-tolerant wide-area application infrastructure," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 1, January 2002.
- [22] M. L. Curry, R. Klundt, and H. L. Ward, "Using the Sirocco file System for high-Bandwidth Checkpoints," Sandia National Laboratories, Technical Report SAND2012-1087, 2012.