Massively Parallel Loading

Wolfgang Frings
Jülich Supercomputing Centre
Forschungszentrum Jülich
52425 Jülich, Germany
w.frings@fz-juelich.de

Dong H. Ahn
Lawrence Livermore National Laboratory
Computation Directorate
Livermore, CA 94550
ahn1@llnl.gov

Matthew LeGendre
Lawrence Livermore National Laboratory
Computation Directorate
Livermore, CA 94550
legendre1@llnl.gov

Todd Gamblin
Lawrence Livermore National Laboratory
Computation Directorate
Livermore, CA 94550
tgamblin@llnl.gov

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Computation Directorate
Livermore, CA 94550
bronis@llnl.gov

Felix Wolf
German Research School for Simulation Sciences
52062 Aachen, Germany
RWTH Aachen University
52056 Aachen, Germany
f.wolf@grs-sim.de

ABSTRACT

Dynamic linking has many advantages for managing large code bases, but dynamically linked applications have not typically scaled well on high performance computing systems. Splitting a monolithic executable into many dynamic shared object (DSO) files decreases compile time for large codes, reduces runtime memory requirements by allowing modules to be loaded and unloaded as needed, and allows common DSOs to be shared among many executables. However, launching an executable that depends on many DSOs causes a flood of file system operations at program start-up, when each process in the parallel application loads its dependencies. At large scales, this operation has an effect similar to a site-wide denial-of-service attack, as even large parallel file systems struggle to service so many simultaneous requests. In this paper, we present SPINDLE, a novel approach to parallel loading that coordinates simultaneous file system operations with a scalable network of cache server processes. Our approach is transparent to user applications. We extend the GNU loader, which is used in Linux as well as proprietary operating systems, to limit the number of simultaneous file system operations, quickly loading DSOs without thrashing the file system. Our experiments show that our prototype implementation has a low overhead and increases the scalability of Pynamic, a benchmark that stresses the dynamic loader, by a factor of 20.

Categories and Subject Descriptors

D.4.9 [**Software**]: Operating Systems—*Systems Programs and Utilities*

Keywords

HPC; Scalability; OS/runtime service

1. INTRODUCTION

Software complexity is increasing in high performance computing (HPC) applications, and dynamic linking and loading [13] offer advantages for managing this complexity. Dynamic libraries allow large applications to be modularized, or split into independently built packages. Modules can greatly reduce build time for developers, and each module can be loaded and unloaded dynamically at runtime as needed. This saves memory and frees developers from the burden of maintaining multiple pre-configured builds of large applications. Further, dynamic loading is used extensively by dynamic languages such as Python. High-level abstractions in these languages enable the rapid development of new plugins and packages, and they reduce software complexity for application programmers. These features are increasingly used to manage complexity in large, multi-physics simulation codes, which may have millions of lines of code and thousands of runtime-configurable parameters.

Despite these benefits, there are serious scalability problems with most modern loading mechanisms. Dynamic linking defers the process of locating object code and resolving its symbols until runtime. To load a library, most loaders first search a list of file system locations until the target library is found, then they load the library into memory. This mechanism does not differ significantly from the method used by the MULTICS [6] operating system that introduced dynamic linking in 1964. While it was suitable for a single-node machine, it breaks down in parallel. Jobs on today's largest machines [12] may have over a million concurrent processes, and concurrency is generally increasing. In practice, most processes load the same libraries they depend on simultaneously. Modern supercomputers typically lack node-local storage, and even large parallel file systems cannot quickly service millions or billions of small, simultaneous I/O requests. Loading an application initiates an I/O storm that manifests much like a denial-of-service attack. The file system, often shared site-wide, becomes unavailable, and dependent jobs sit idle until all requests are cleared.

The dynamic loader has remained much the same for over four decades, but it must change to reach extreme scale. To load object code on systems with potentially billions of processors, the loading service itself must be made parallel, and it must efficiently coordinate trillions of concurrent load requests. In this paper, we take advantage of the inherent parallelism of large-scale loading. In most cases, object code is read-only, and the same code is requested simultaneously by many processes. Parallel loading is very similar to a broadcast. We have developed Scalable Parallel Input Network for Dynamic Loading Environments (SPINDLE), a novel solution that extends the dynamic loader to route load operations through a network of scalable file-cache servers. Only a small set of designated daemons communicate directly with the file system, and SPINDLE proactively distributes and caches DSOs.

This paper makes the following contributions:

- Detailed analysis of real-world impact of scalability problems with today's dynamic loader;
- Novel techniques to transparently coordinate dynamic loaders through an existing interface;
- An architectural blueprint for a global parallel loading service;
- A portable implementation of this architecture including integration with portable software infrastructure.

We have applied SPINDLE to Pynamic, a parallel benchmark that stresses dynamic loading performance. Using SPINDLE, Pynamic can run on 20 times as many processes without overloading the file system. Further, the performance overhead of our scheme is constant with respect to the number of processes. Overall, our evaluation suggests that our approach to dynamic loading offers a viable path to massively scalable loading on extreme-scale systems.

The remainder of the paper is organized as follows. Section 2 motivates our work and reviews the state-of-the-art techniques. Section 3 then describes the software architecture and key elements of SPINDLE. Section 4 details our experimental evaluations, and Section 5 summarizes our findings and discusses future work.

2. SCALING CHALLENGES

The dynamic loader, also called the dynamic linker, is responsible for locating object code and making it available within a process's address space. These actions make the object code's subroutines callable by the main program. Dynamic linking and loading first appeared in the MULTICS [6] operating system (OS) in 1964, as a way to share common code between processes. For this reason, dynamically linked libraries are also often called *shared libraries*, and any object files that can be linked and loaded by the dynamic loader are called *dynamic shared objects* (DSOs). Most HPC systems including Linux clusters, IBM Blue Gene, and Cray machines use the dynamic loader implementation from GNU's libc, which is based on standards detailed in the System V Application Binary Interface (ABI) [1].

The dynamic loader is implemented as a DSO that is loaded by the OS during process start-up. The OS gives control to the loader, which then loads the main executable's dependent libraries and transfers control to the executable's entry point. The executable may re-invoke routines in the dynamic loader to resolve symbols or to load new DSOs during runtime (via routines such as `dlopen`). The tasks of finding and loading object code involve many file system operations, and the simultaneous execution of many such operations is not scalable. Loading massively parallel applications thus has the potential to overwhelm a site-wide shared file system, which can disrupt other applications running across the entire computing facility.

2.1 Scaling Challenges with the KULL Code

KULL [15] is a large multi-physics application developed at LLNL. When it was first run on Dawn, an IBM Blue Gene/P system, its start-up exposed serious scaling challenges and significantly disrupted the entire computing center. The KULL executable was dynamically linked to about one thousand shared libraries, most of them staged in an NFS file system to ease maintenance. Loading these libraries during program start-up scaled *extremely* poorly. The time from job launch to the initial invocation of KULL's `main` function, for instance, took about one hour at 2,048 MPI processes and *ten* hours at 16,384 processes. Further, during start-up, the NFS server was overwhelmed with load requests from KULL, and other users and jobs were unable to connect to the NFS while KULL start-up continued.

Our initial analysis revealed that the large-scale parallel loading created an enormous I/O storm that far exceeded the file server's capability. In particular, the dynamic loader (`ld.so`) was making massive numbers of unnecessary open calls at scale. On 16,384 MPI processes, we instrumented the loader and determined that it issued 300 million `open` calls to the NFS server. As a first attempt to address the problem, we staged the application executable and its library dependencies on Lustre [17], a parallel file system mounted on this machine. When we saw no improvement, we commenced a more detailed analysis.

2.2 Non-Scalable Loader Algorithm

The dynamic loader performs two types of file system operations: queries to locate a DSO and reads to load its contents into memory. Locating a DSO is necessary because an executable does not specify its dependent libraries with full path information, but instead provides the *names* of the libraries it needs loaded. To load a particular library, the dynamic loader searches for files with its name in system locations (e.g., `/lib`), directories named in the executable (rpaths), or directories named in environment variables such as `LD_LIBRARY_PATH`. The GNU implementation tests for existence by appending the name to each directory and calling `open` on the resulting path.

Search paths are a form of late binding that supports a flexible user environment. By simply modifying the search path, users can run the same executable with an updated or improved version of a library without recompiling. The concept has remained largely unchanged since MULTICS. However, this algorithm requires a large number of file system operations, which are not a problem for a sequential program with a dedicated file system. A parallel application with P processes, L library dependencies, and D directories in its search path will perform $O(PDL)$ file system operations simply to locate shared libraries; this accounted for the bulk of the 300 million `open` calls in our KULL example.

Once a library has been located, the dynamic loader maps it into memory. Each library contains a table of program headers that describe what parts of the library on disk should be mapped into memory. Typically, its code and data are mapped into memory while its debug and linking information are not. Unfortunately, the loading protocol is similarly unscalable. The GNU loader uses `open` and `read` system calls to access the program headers, then `mmap` system calls to load the bulk of the library into memory. A parallel application with the GNU loader will thus perform $O(PL)$ load operations during its start-up.

Nearly all major runtime environments in use today use these algorithms to locate and to load executable code. Languages like Python and Java use search paths to find and to load modules. Major OSs such as Mac OS X and Windows also use search paths in their dynamic loaders. When run in parallel, these approaches produce $O(PDL)$ search operations and $O(PL)$ load operations, and this will thrash the file systems in the scaling limit of P.

2.3 State-of-the-Art Approaches to Loading

We are not the first to address the problem of scalable parallel loading. In this section, we discuss existing solutions to put our work in context. The approaches discussed here can be broken down into two categories: those that attempt to improve I/O and storage technologies so that the existing, unscalable loading algorithm will perform faster, and those that attempt to modify the loader.

2.3.1 Parallel File Systems

Staging object code in parallel file systems [9, 17] so that the loader can access it more effectively seems like a simple solution to our problem. Modern parallel file systems are clusters themselves that combine multiple disks spread across their nodes into a logical unit. While access to a single disk does not scale well, parallel access to the array of disks scales to the number of nodes in the cluster. Thus, parallel file systems like Lustre [17] significantly improve data-parallel access to large data-set files that are striped across the array of disks. Unfortunately, parallel loading does not exhibit this access pattern.

Parallel loading exhibits no data parallelism, as each process accesses the same, small files. Worse, for library search, parallelism is needed for large numbers of *metadata* operations, and parallel file systems typically have far fewer metadata servers than data servers. As a result, our analysis shows that while they offer a performance advantage over NFS when used with a traditional loader, they do not address the fundamental scaling problem in the loader. Coordinating I/O in parallel among the loaders themselves can easily outperform both Lustre and NFS.

2.3.2 Caching and Staging Solutions

Many large-scale systems, such as IBM's Blue Gene machines and more recent Cray XT machines, have dedicated I/O nodes that sit closer to the compute nodes than the parallel file system. A common approach to loading on these architectures stages object code on these nodes and mounts the staging area on the compute nodes. This approach has been used at Argonne National Laboratory (ANL) to accelerate the loading of Python applications on the Intrepid Blue Gene/P machine. It is effective in speeding up the loading process, but it is not transparent to users.

First, it requires application developers to stage their application in a custom I/O node image, which can be tedious when there are large numbers of libraries. Second, users often cannot easily determine which libraries an application needs to load. Most end-users of Python, for example, are not familiar with its standard libraries or with those that have DSOs which need to be staged. Third, in some cases the library search path is not known until runtime, so it is impossible to stage *all* loads that the application performs.

Cray uses DVS [10, 11], a proprietary I/O forwarding service, to make this process more transparent. DVS dedicates an NFS server to compute nodes on which it attempts to aggressively populate a hierarchy of caches along the I/O forwarding path. However, this approach requires users to stage all application-shared libraries to the dedicated NFS server, which can be tedious — similar to ANL's approach. Further, they do not exploit the full available parallelism of read-only dynamic load routines. Caching directly in the loader is a more direct, coordinated and scalable approach.

2.3.3 A Peer-to-Peer Solution

Dosanjh, et al. propose a peer-to-peer (P2P) architecture for distributing shared libraries across a network [7]. The approach is similar in spirit to our own, in that it caches loaded libraries in a RAM disk and aims to satisfy most load requests within the HPC network to reduce file system load. The approach has the potential for high-bandwidth file distribution, as it is based on the BitTorrent protocol. The authors share our vision of a high-availability parallel loading service; their architecture integrates a loading daemon with the OS. It is difficult to draw a fair comparison with this work, because it is still in early design stages.

As proposed, the P2P approach does not address two key problems that SPINDLE addresses. First, the authors' design requires users to specify all library dependencies in a job description so that they can be seeded to the compute nodes for P2P sharing. This requires, as do the techniques in Sections 2.3.1 and 2.3.2, that users know all library dependencies of their application and specify them in advance. SPINDLE optimizes the case where dependencies are known in advance, but it is still efficient when dependencies are not known until runtime. Second, the proposed approach does not address the metadata storm resulting from large-scale application start-up. The authors mention that initial seeder processes must handle the first set of requests for libraries as well as a large number of `stat()` calls issued by the dynamic loader. However, they do not discuss a coordinated I/O strategy that would allow these seeders to satisfy millions of requests quickly for each job launch. The authors mention distributed hashing techniques in the paper, which are promising for scalable P2P loading. We have addressed these problems in SPINDLE with a low-latency tree-based architecture, and by using the rtld-audit interface to modify the loader's behavior. Our approach does not require OS daemons and runs entirely in user space.

2.3.4 Solutions that Modify the Loader

The `collfs` library [4] developed jointly by ANL and the King Abdullah University of Science and Technology (KAUST) provides a scalable dynamic loading service for Python applications. `collfs` allows one process to load libraries that it broadcasts synchronously to the full system. This solution customizes the GNU loader and the Python runtime to use MPI collectives to load libraries. This approach is effective and can drastically speed up many Python programs, but the implementation changes the semantics of loading by requiring that it be synchronous. When used this way, some Python programs will require modification so that all loads are performed at the same time, otherwise programs may deadlock.

Our solution handles asynchronous loads of different libraries as well as synchronous loads of the same DSO. `collfs` is a good example of how coordinated I/O can speed up loading, but its synchronous semantics and application-specific nature limit its use. It also has technical limitations for system-wide use: it requires a modified version of `glibc`, which makes it very difficult to install. In contrast, SPINDLE uses the rtld-audit interface, which allows us to intercept GNU loader actions without requiring direct modifications to the loader itself. `collfs` also relies on the MPI library, which is not always available at runtime. Our solution works with any programming model for large-scale systems.

2.4 Loading as a Parallel Service

Parallel loading is an example of a case in which a sequential solution applied by each of P processes does not yield good parallel performance. Our key observation is that most processes request the *same* objects from the file system, or at least objects that have also been loaded by nearby processes. Rather than accessing the remote file system each time a file is needed, we should exploit the likelihood that a neighbor has already requested the file. Thus, we can coordinate I/O to distribute files much more efficiently and, thus, reduce the stress on the limited file system resources.

Alternative techniques such as DVS (discussed above) attempt to increase the level of I/O coordination transparently at the file system level while keeping loader behavior fixed. These techniques have the advantage of maintaining existing abstractions, such as POSIX I/O, which are familiar to users of UNIX-like OSs and which make sense in a sequential environment. However, the strict semantics of such abstractions can limit their scalability in a parallel environment. As an example, POSIX file I/O semantics disallow caching of failed `open` calls, forcing every library search query to go all the way to the file system. Further, traditional file I/O abstractions are oblivious to the *type* of data being transferred, which precludes many parallel optimizations.

We raise the abstraction to the level of the loader, which allows the loader to perform its own coordinated I/O. Thus, we can exploit knowledge about the files in parallel use. Object code is nearly always read-only and further ample parallelism exists in parallel loading. The performance advantages of exploiting both of these characteristics are too great to be ignored in a parallel environment. For this reason, we recommend that the loader architecture be changed for parallel machines. SPINDLE represents a significant step towards such a truly massively parallel loading service architecture.

3. THE SPINDLE APPROACH

Parallel applications that follow the SPMD model, which is the standard in modern HPC, issue duplicate load requests across a potentially large number of processes. On large-scale systems, even applications that employ the MPMD model tend to divide processes into SPMD groups. Thus, most load requests occur at program start-up and are concentrated in a very short interval to a common set of files, which causes the type of file-access storm that we described previously. However, we can also exploit this temporal locality to solve the problem. SPINDLE intercepts requests for libraries made by the dynamic loader, and it services them with a combination of scalable parallel broadcasts and a local file-system cache. The cache provides efficient local access, and broadcast operations allow each requested object to be loaded from the file system only once, then propagated scalably to the nodes that need it. This removes the $O(P)$ scaling bottleneck that we discussed in Section 2.2.

In addition to temporal locality, parallel loading exhibits spatial locality. With high probability, each process loads the same sequence of libraries at start-up, because link dependencies are embedded in the executable and libraries. While we do not *require* this strong locality of reference for efficient loading, our SPINDLE implementation can exploit it.

The SPINDLE broadcast operation can be initiated in two different ways. With the *push* model, we assume that when SPINDLE receives the first request for a particular library, other processes are likely making the same request. With *push*, SPINDLE immediately broadcasts each library to all processes when it is first requested. In contrast, the *pull* model sends libraries to nodes only as they request them. The *push* model is more efficient for SPMD applications where each process needs the same DSOs. However, it can waste memory in MPMD applications by pushing libraries onto processes that do not require them. The *pull* model, which we are still developing and testing, will be more appropriate in MPMD applications.

Figure 1 shows SPINDLE's architecture for coordinating the loading procedure among different processes of a parallel application. It maintains a collection of load servers (daemons) alongside the application processes, which cache replicas of DSOs in local RAM disks. A client adapter dynamically linked into each application process redirects all load requests to a nearby load server. In this way, we keep the client adapter as lightweight as possible. To combine load requests for the same object issued by different application processes, and to facilitate efficient broadcasts, the servers are attached to a network that connects them to each other and to the underlying file system.

3.1 Overall Architecture

As mentioned, application processes usually load most of their required libraries during start-up. However, MPI and other runtime communication systems are typically unavailable during this time. That makes it difficult to coordinate loading in parallel. This restriction leads to our first design decision: SPINDLE establishes its own independent communication network, implemented as an overlay network on top of the existing one. The load servers are designed to communicate with an arbitrary number of application processes and other load servers at the same time. This provides the flexibility needed to build customized overlay network topologies that optimally match the underlying hardware communication structure and also the capabilities of the file systems. While Figure 1 shows a tree topology with one root server connected to the file system, SPINDLE can configure more efficient topologies such as a forest of trees whereby multiple roots connect to a more capable file system. Further, the daemon concept keeps our design open to an integration with other system services, thus forming the precursor of a more general parallel loading service architecture.

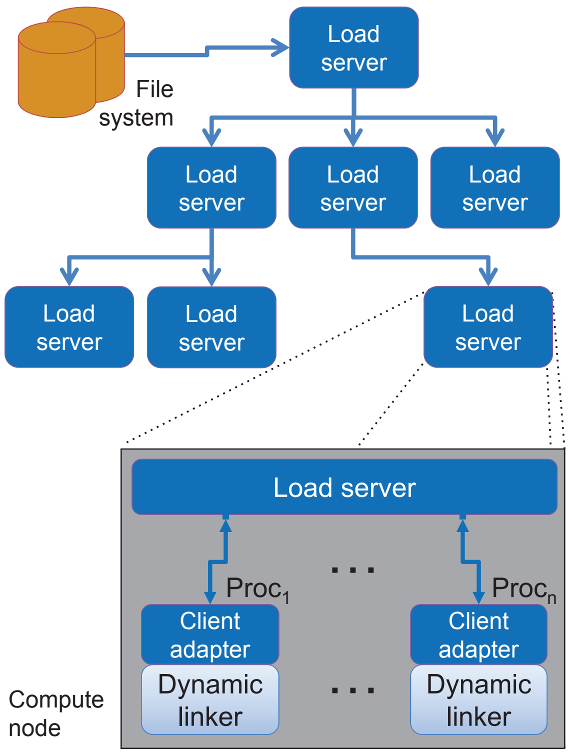


Figure 1: Overall architecture of SPINDLE

Finally, the application client adapter has to reroute all file-system requests such as searching for libraries or loading the library code to the SPINDLE servers. To avoid implementing our own dynamic loader, we decided to use the rtld-audit auditing API of the GNU loader for this purpose. This interface allows user code to intercept load requests and modify the dynamic loader's behavior. Our design allows us to run all components of SPINDLE in user space without having to modify the runtime system. Below, we explain the components of SPINDLE including the client, the server, the overlay network, and the motivation behind their design.

3.2 Client Adapter

The client adapter hooks into the dynamic loader, intercepts load requests and file searches, and redirects them to a nearby load server. To implement the interception, we use the rtld-audit [16] interface provided by the GNU dynamic loader. The interface allows us to register callbacks that are invoked when the dynamic loader performs certain operations. One of these callbacks triggers before the loader tests a directory for the existence of a library. This callback can optionally return an alternate location that the loader should look in. Our implementation of this function reroutes load requests to the SPINDLE load server, which obtains the DSO and returns its location in the local RAM disk. Our callback then returns this location to the loader for access from the faster local storage.

The rtld-audit interface was originally intended to help debug the dynamic loader, thus enabling it has several implications when used for our purpose. First, the callbacks for rtld-audit are supplied in a DSO, which is specified via the `LD_AUDIT` environment variable, and loaded into a special library namespace. While this namespace protects the rtld-audit library by disallowing other application libraries from seeing its symbols, the strong isolation also restricts its access to functionalities in other libraries, with an exception for libc. This restriction prevents the client adapter from using the TCP/IP stack, which resides in additional dynamic libraries. Instead, our adapters use UNIX-named pipes to communicate with their load servers, which can be managed directly using libc. Because named pipes do not allow inter-node communication, our design currently runs at least one load server on each node.

Second, enabling rtld-audit puts the dynamic loader into debug mode and has a performance impact on the application. Under normal execution mode, the first time an inter-library call-site is invoked, the dynamic loader will look up its target function and cache it in the global offset table (GOT). Subsequent invocations of the call-site will use the GOT entry and skip the lookup. This binding has performance advantages, in particular when this call-site is executed multiple times. With rtld-audit enabled, however, the loader will not finalize the binding by filling in the GOT, which allows rtld-audit callbacks to be delivered upon every inter-library call. This can impose an additional performance overhead, which we address by having the client adapter take over the responsibility of filling in the GOT for the dynamic loader.

Third, the rtld-audit interface does not provide any mechanism for intercepting loading of the executable or rtld-audit library. However, loading the executable through SPINDLE is particularly important because executables can contain a large percent of application code and cause significant load on shared file systems. SPINDLE solves this using LaunchMON [2], a scalable tools infrastructure component for launching parallel applications and tools across a wide range of HPC platforms. The user gives SPINDLE the command that launches the parallel application. SPINDLE then modifies it to launch a small statically-linked bootstrapper executable first and transfers control to LaunchMON. Next, LaunchMON starts this bootstrapper alongside the load server on each node. Finally, the bootstrapper works with the load server to move the executable and rtld-audit library into the RAM disk and then `exec`'s the application.

In addition to DSOs, SPINDLE can accelerate the parallel reading of Python modules. Interpreted languages like Python load not only DSOs, but also the script and bytecode files required by external modules. These can often be more numerous than shared libraries. When the Python interpreter `open`s such a file, our client adapter intercepts the attempt using another rtld-audit callback function, which is called when an entry in the procedure linkage table (PLT) is resolved. We route these files through the same caching mechanism: they are stored on the local RAM disk and the `open` call is rerouted to their new location.

3.3 Load Server

To handle DSO search and load requests for clients, the load server plays three roles: it manages local replicas, interacts with other load servers through an overlay network, and accesses the file system if designated as a reader. Locally, the server stores copies of DSOs and metadata on search directories and libraries. The files are stored in a RAM disk, and the server maintains the metadata in a hash table.

3.3.1 Operation

File existence tests contribute to a significant part of the shared file system load, and the SPINDLE server caches the contents of directories to optimize these operations. When the dynamic loader tests a file path for existence, the operation is rerouted to the load server. The server looks up the corresponding directory in its metadata table. If it exists, the table provides information on all files in the directory and the existence of a file can be verified locally. Otherwise, the load server triggers a designated-reader protocol, whereby a server responsible for the directory reads it via file-system operations and broadcasts the results to the rest of the servers. Similarly for file data, one server reads the file from the file system and distributes it to the remaining servers, which store it on their local RAM disks.

In terms of data management, we use *replication* schemes, where all objects are proactively replicated across servers, as opposed to *distribution* schemes where data distributed among the servers reactively services requests. Although replication requires more memory, we have made this design choice because it offers a significant performance advantage over the alternative. The load requests typically occur in the same order in a short interval, and thus our replication-based *push* model can better exploit such strong locality of reference. For example, the replicated metadata allow the bulk of search operations to be satisfied locally without relying on frequent server-to-server communication. Library files themselves will also be locally available, shortly after one server reads the file from the file system, ready to service other identical requests anticipated shortly.

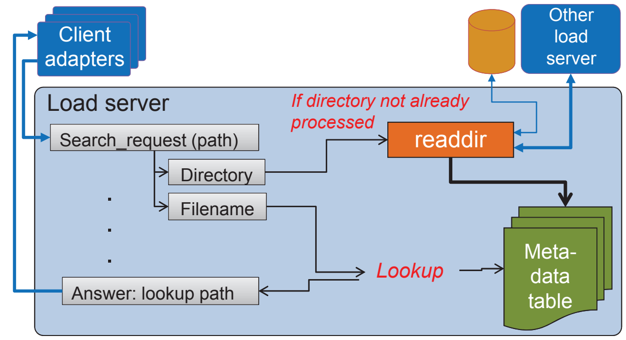
Figure 2 shows the implementation details as to how the load server quickly replicates file data as well as metadata. Figure 2(a) illustrates the metadata handling: on a search request, a server either parses the directory by itself or forwards the search request to another nearby server in the overlay network. Figure 2(b) shows the file data: one reads the file from the file system, stores it on its local RAM disk, and distributes it to the remaining servers. These are enough to trigger our designated reader scheme.

3.3.2 Memory Overhead

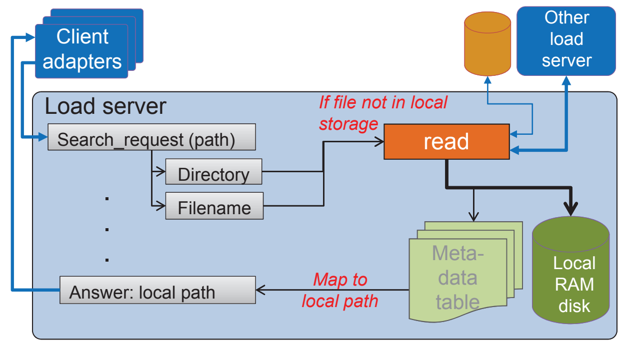
While SPINDLE improves the performance and scalability of shared library loading, it does this at the expense of using extra memory on each node. The amount used is approximately equal to the total amount of code in libraries and the executable minus the amount of code in the application's working set. In other words, SPINDLE makes an application use memory as if every code page were in its working set.

During normal execution, Linux loads code pages into physical memory only when they are first touched. Under SPINDLE, all code is stored in a RAM disk and kept in physical memory for the entire execution of an application. The RAM disk is not a direct duplication of code pages—Linux is smart enough to use the same physical memory to back both the RAM disk and the application's virtual memory pages. The bulk of SPINDLE's memory overhead thus comes from placing code pages into the RAM disk that would not have been touched or loaded without SPINDLE.

While code preloading incurs extra memory overhead, it has advantages for extreme-scale environments. Loading pages from disk as-needed during execution is a well-known source of undesirable OS noise. Thus, high-end systems like



(a) Meta data lookup operation



(b) File read operation

Figure 2: Load requests handling in load server

IBM Blue Gene intentionally preload pages into memory and avoid this noise [8].

As an optimization, SPINDLE does not load the entire library file into RAM disk. Each DSO specifies parts of itself that should be loaded into memory via a table of *program headers*. Typically, a DSO will specify that its code and data should be mapped into memory, while debug information and the symbol table are left on disk. The load server reads these program headers and caches only the parts of a library that are needed at runtime. Essentially, SPINDLE strips libraries before transmitting them. This does make using a debugger on a SPINDLE application difficult, and we have future plans to hide these effects by pointing debuggers back to the original library files.

Memory used by libraries loaded by SPINDLE is treated differently in low-memory situations. Since the memory pages that store libraries are backed by a RAM disk, they cannot be paged out when the system runs low on memory. If those pages were backed by a shared file system, they could be dropped and reloaded as needed. This means an application using SPINDLE may run out of memory if it uses paging without SPINDLE.

3.4 Overlay Network

To funnel requests to the file system and to broadcast DSOs back to the application, SPINDLE load servers have to rely on a communication infrastructure. In the absence of an application runtime communication system like MPI, at start-up we have to create our own overlay network on top of the system network infrastructure.

For this, we use COBO [5], the Collective Bootstrapper, a scalable implementation of the PMGR collective protocol that is used as an MPI job start-up mechanism. Unlike the original PMGR protocol, which connects all clients to one master client, COBO uses a scalable tree topology. COBO is part of the LaunchMON software infrastructure, which allows tool servers to be co-located with a parallel job. As discussed before, SPINDLE integrates LaunchMON to start the load servers along with the application processes.

The overlay network is initialized during application start-up. Given a list of hosts as input, COBO establishes the overlay network by distributing connection information from the root down to the leaf nodes. Given that the servers—in contrast to the clients—are not part of the parallel application and are not restricted to libc functions, we can use TCP/IP sockets for inter-server communications. COBO provides collective communication operations, which we use to distribute libraries from the root server to all others.

In the following, we present two network topologies: a single tree as our base configuration and a forest to exploit the bandwidth of parallel file systems. We also discuss bulk preloading, an optimization that anticipates load requests even earlier based on static analysis of the executable.

3.4.1 Single Tree

To connect n nodes, COBO creates a binomial tree of degree d ($n < 2^d$). According to the properties of such a tree, messages broadcast from the root will arrive at each node after not more than d hops. Thus, tree depth and walk distance vary no more than logarithmically with the number of load servers.

3.4.2 Forest

For larger configurations, we also offer the option of a forest consisting of multiple trees. In this way, we can take advantage of parallel file systems, reading file data from a moderate number of root nodes concurrently. Assuming that these root nodes read their data in parallel, the propagation of data inside individual trees is also accelerated. For example, when using d trees, the size of individual trees will be reduced by a factor of 2 in comparison to a single tree and the maximum distance will reduced to $d-1$ hops. In the current implementation, the forest is built by dividing the host list into different partitions and deploying a separate overlay network for each of these partitions.

3.4.3 Bulk Preloading

SPINDLE normally loads a library if at least one of the processes requests it at runtime. A complementary approach is to predict the demand in advance and stage all required libraries in advance. SPINDLE statically analyzes the executable and extracts its library dependencies. With these, it can initialize the caches of the load servers without waiting for runtime load requests. The subsequent requests for the preloaded files are fulfilled immediately. Bulk preloading does not have to cover every possible library. Libraries accessed dynamically, such as those loaded via `dlopen`, can still be loaded through SPINDLE's push mechanism. In the future other hints, such as suggestions from the user, could be used to increase the number of preloaded DSOs.

SPINDLE provides bulk preloading through the LaunchMON front-end client. LaunchMON replaces the normal start-up command such as `mpiexec` and starts the parallel application along with the tool servers. Since the front-end client is also part of the overlay network, it enables direct communication with our load servers. This allows us to proactively push a collection of libraries to the load servers before the application requests them. On HPC systems where the runtime environment differs between front- and back-end nodes, the paths of preloaded library have to be changed to the back-end location. For this, we use FGFS [3], a scalable utility to obtain global file properties.

3.5 Caching Algorithms

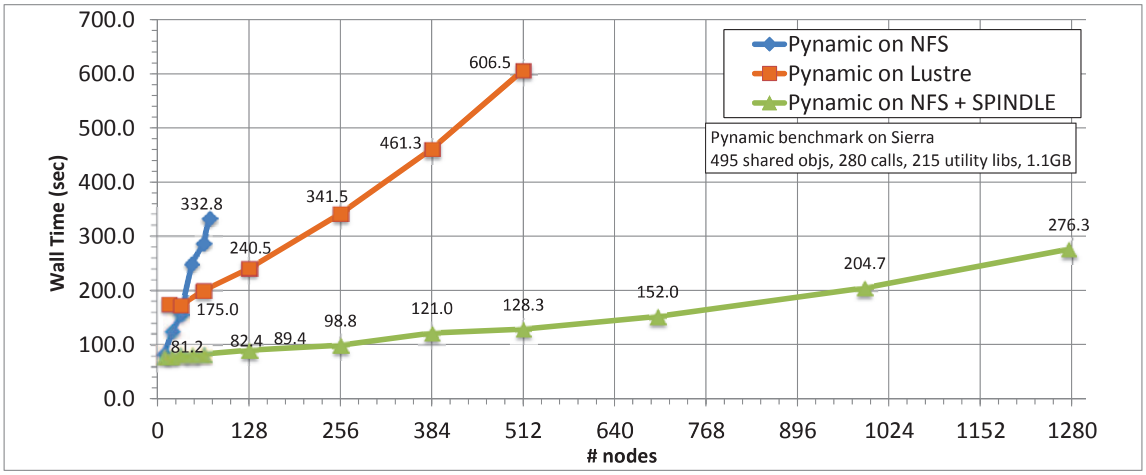
Our design is flexible enough to support a variety of algorithms for request handling, caching, and information forwarding. In its most general form, search and read requests are injected into the server network by a client. The server forwards the request to another server that is designated as responsible for the desired file. An internal mapping function decides which server this is. The server returned by this function then performs the actual file operation on behalf of the client and broadcasts the file data to all remaining servers. Changing the mapping function is one way of configuring SPINDLE's behavior.

In the current implementation, the network topology is a tree or a forest. This constrains the communication scheme to a top-down distribution of information from the root node to other nodes. (Figure 1). The mapping function in this case is quite simple: the root server is responsible for all library files. Viewed from the perspective of the file system, the load a parallel application generates is equal to the load of a single process times the number of trees in the forest, which is a configurable number and usually small. Our initial implementation assumes that all processes request the same libraries in the same order, which is not uncommon for SPMD codes. In this case, a load server does not have to forward an incoming request up the tree. Since the root server will receive all possible requests from its local client, an arbitrary server just has to wait until the root server pushes the desired libraries in its direction. This policy reflects a proactive top-down distribution of cache data, but can cause problems if different processes load different libraries. This assumption eased our initial implementation and provided convenient optimizations, but it is not fundamental to the SPINDLE approach.

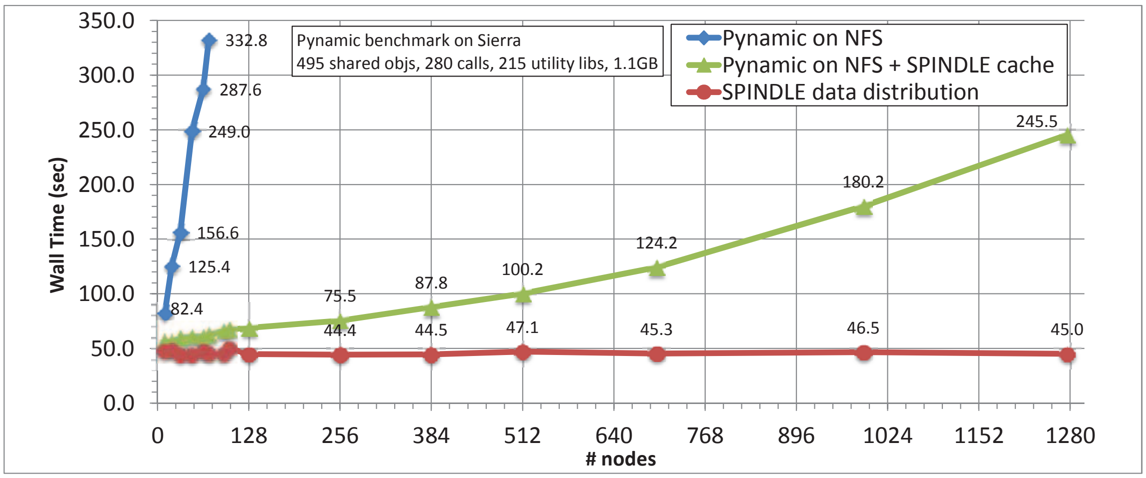
In the future, we plan to provide more sophisticated file-distribution schemes. Using point-to-point communication between servers, a function we already added to COBO, can provide an on-demand distribution scheme where each server can actively inject search requests into the network and maintain full authority over the contents of its local cache. This will allow us to combine the *pull* model with the *push* model. Further optimizations include the automatic reduction of data exchanged among load servers. For example, we could check whether a requested library is a system library. System libraries may reside in a node-local file system, where caching is not needed. Finally, we plan to improve server efficiency through multithreading to allow local and network operations to be performed concurrently.

4. EVALUATION

The main purpose of SPINDLE is to reduce the start-up time for parallel applications that extensively use DSOs. To verify this and also to show the scalability of our approach, we tested SPINDLE with a synthetic benchmark named Pynamic on the Sierra Linux cluster installed at LLNL. In the



(a) NFS and Lustre file systems vs. NFS under Spindle



(b) Using bulk preloading

Figure 3: Pynamic benchmark run on LLNL Linux cluster Sierra

following, we will introduce the Pynamic benchmark, show the results of Pynamic running without and with Spindle support for the Sierra system up to 1,276 nodes, and discuss the memory footprint of Spindle.

4.1 Pynamic Benchmark

Pynamic is a benchmark that supports configurable emulation of dynamic loading in Python-based applications on massively parallel systems. This pyMPI-based [14] benchmark allows users to configure and generate arbitrary numbers of Python dynamic modules and utility libraries of a arbitrary sizes. It uses a code generator to create these DSOs, which are then used by a dummy application to closely model the behavior of Python-based multi-physics applications. Pynamic outputs three performance metrics that capture three phases of real-world dynamic loading and linking: start-up time for the initial library loading; module-import time for symbol resolution; and visit time for execution. Because dynamic loaders typically provide options that can shift these overheads from one phase to another, the sum of the three metrics is the overall figure of merit.

4.2 Results on Sierra

We ran our tests on the Sierra Linux cluster at LLNL, which is equipped with 1,856 compute nodes. The nodes have a 2-socket Intel Xeon EP X5660 CPU (2.8 GHz) with 12 cores and 24 GB of RAM. Nodes are connected by a Qlogic Infiniband QDR interconnect. We used two systems: NFS and Lustre. On Sierra, NFS is used for the home file system and Lustre is used for scratch data space. In all of these tests, we configured Pynamic to load 495 shared objects, a total of 1.1 GB of library files.

For our first test, we ran Pynamic without Spindle and loaded libraries from NFS. As shown in Figure 3(a) the overall runtime increased rapidly. We stopped this measurement at a small scale ($<$ 100 nodes) to prevent an I/O storm that would affect other users on the system. The run-time increased exponentially, which can be explained by the poor parallel support of NFS.

As described in Section 2.3.1, a parallel file system is better suited to the types of file operations used in Pynamic. Therefore, we ran the same test on Sierra with the library files staged on Lustre. Figure 3(a) shows more linear growth in the runtime under Lustre, and we ran the test up to 512 nodes, 6,144 processes. The linear scaling of these runs shows significant improvement compared to prior experiences with Lustre (which had suggested that NFS might perform better than Lustre). A likely cause was the relatively recent enabling of read caches on the Lustre servers. We would expect further improvements if Lustre were better configured for the type of I/O access pattern associated with Pynamic (small files and high metadata rate), though this is not the typical file access pattern for HPC file systems (large files and data parallelism). Our measurements show that Lustre could be a partial solution to the loading problem, but only up to a moderate number of processes. The Lustre performance was already starting to degrade at these scale, and would likely have suffered more at larger scales.

In our third test case, we ran Pynamic with SPINDLE, with libraries hosted on NFS. As shown in Figure 3(a), SPINDLE allows us to run Pynamic at the largest allocation size on Sierra, with 15,312 processes distributed over 1,276 compute nodes. SPINDLE reduced the overall Pynamic runtime to 276 seconds. This is faster than the NFS test with 64 nodes and the Lustre test with 256 nodes.

Though SPINDLE allowed us to run Pynamic at a significantly larger scale, we still saw unexpected growth in the total runtime, from 81 seconds at a small scale to 276 seconds at higher scales. Pynamic's built-in timings did not help us to distinguish whether this time increase was due to poor scaling in SPINDLE or whether it was part of Pynamic's reliance on PyMPI (which was known to have scalability issues). To get a better understanding of this effect, we designed a further benchmark run where the library load process was separated from the benchmark execution. We enabled the bulk-preloading feature of SPINDLE (see Section 3.4.3) where the front-end process reads the list of prerequisite libraries and populates the load server caches before Pynamic starts.

This allowed us to measure the timing for SPINDLE data distribution, but removed from the Pynamic overhead. Figure 3(b) shows the results. Bulk-preloading takes constant time (though in theory it should be logarithmic at large enough scales). The larger SPINDLE scale tests were performed with multiple COBO trees (see Section 3.4.2), which limited the depth of each individual tree and the time to broadcast data. The largest test was done with four trees, each with depth eight. Given that the total Pynamic runtime continued to grow, even with all files pre-staged to the RAM disk, we concluded that this growth came from Pynamic's communication rather than library I/O. We will work with the Pynamic developers to eliminate these scaling problems in a new version of the benchmark.

Given the flat scaling of the data distribution phase, we expect that SPINDLE will continue performing well on future systems and it represents a viable path towards dynamic loading at exascale.

4.3 Memory Usage

As discussed in Section 3.3.2, the SPINDLE memory overhead is dependent on the percent of code pages that an application normally maps into memory. SPINDLE causes the application to use memory as if 100% of its code pages were always resident. We measured this overhead on a single node of Sierra using Pynamic, configured to load 495 libraries, which produced an application with 488 MB of loadable code. We modified this version of Pynamic to touch a certain percent of its memory pages artificially, loading them into its working set. This allowed us to compare SPINDLE memory overheads for different working set sizes. The amount of memory used was measured by running Pynamic, touching the pages and then checking how much remaining memory could be allocated before the system began paging out to make room for the new allocations.

Figure 4 illustrates two aspects of our memory overhead. First, it shows that the memory overhead is predominately a factor of the application's working set size. In the worst cases, where the application only has a small working set, SPINDLE has loaded many unneeded pages into memory via the RAM disk and produced overheads approximately equal to the application's size. In the best cases, where the application needed most of its code, the RAM disk shares most of its physical memory with the process and the amount of memory available does not change significantly. Second, Figure 4 shows that the memory overhead of the client and server is relatively small. The data point at 100% represents the case where the RAM disk memory is completely shared with the virtual memory of the process. The remaining overhead of 15.2 MB is approximately equal to the memory used by the client and the server.

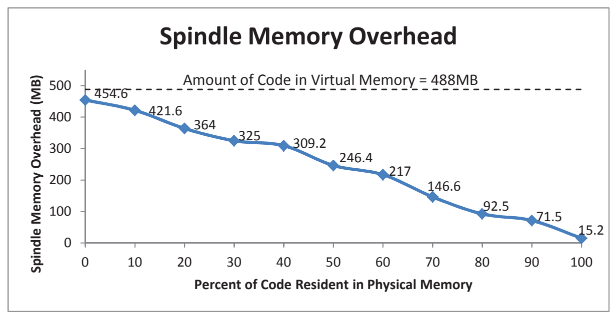


Figure 4: SPINDLE memory overhead

5. CONCLUSION

Dynamic linking and loading have gained broad acceptance in HPC due to their advantages for managing growing application complexity. However, the mechanisms used to load shared objects today do not scale to the level required by the largest supercomputers. The tasks of locating and loading a dynamic shared object involve many file system operations, and when a large number of processes load many DSOs simultaneously, the resulting I/O storm can cripple even the largest parallel file systems. This manifests much like a denial-of-service-attack and can disrupt an entire computing facility. SPINDLE addresses these critical challenges by extending the dynamic loader to coordinate its file system operations with an overlay network of file-cache servers. It is transparent to applications and does not require users to make changes to their applications or workflow. Our Pynamic experiments show that SPINDLE is highly scalable with limited memory and performance overhead.

SPINDLE is already capable of addressing scalability challenges in our production environments with no apparent scalability limit in sight. Thus, what lies ahead of us are mostly engineering efforts to port, optimize, and customize our solution to a broader range of high-end systems including IBM Blue Gene machines with millions of cores. Already, SPINDLE is well positioned for these efforts, as it is an integrated, portable code base. Perhaps more importantly, this basic architecture and our future work will pave the way for a massively parallel OS/runtime loading service for future exascale machines.

6. ACKNOWLEDGMENTS

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. (LLNL-CONF-610893). Part of the compute time used in our experiments was provided by the Jülich Supercomputing Centre.

7. REFERENCES

[1] System V application binary interface. http://refspecs.linuxbase.org/elf/gabi41.pdf.

[2] D. H. Ahn, D. C. Arnold, B. R. D. Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming scalability challenges for tool daemon launching. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, pages 578–585, 2008.

[3] D. H. Ahn, M. J. Brim, B. R. de Supinski, T. Gamblin, G. L. Lee, M. P. LeGendre, B. P. Miller, A. Moody, and M. Schulz. Efficient and scalable retrieval techniques for global file properties. In *The International Parallel and Distributed Processing Symposium*, Boston, MA, 2013.

[4] J. Brown, W. Scullin, and A. Ahmadia. Solving the import problem: Scalable dynamic loading network file systems. Talk at SciPy conference, Austin, Texas, July 2012. http://pyvideo.org/video/1201/solving-the-import-problem-scalable-dynamic-load.

[5] PMGR collective. http://sourceforge.net/projects/pmgrcollective/ (visited April 2013).

[6] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5):306–312, May 1968.

[7] M. G. F. Dosanjh, P. G. Bridges, S. M. Kelly, and J. H. Laros III. A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems. In *Fifth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2012.

[8] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –10, Nov. 2010.

[9] IBM. General Parallel File System. `http://www-03.ibm.com/systems/software/gpfs/` (visited April 2013).

[10] G. Johansen and B. Mauzy. Cray XT programming environment's implementation of dynamic shared libraries. In *Cray User Group*, Atlanta, Georgia, May 2009.

[11] S. M. Kelly, R. Klundt, and J. H. Laros III. Shared libraries on a capability class computer. In *Cray User Group*, Fairbanks, Alaska, May 2011.

[12] Lawrence Livermore National Laboratory. Advanced simulation and computing Sequoia. `https://asc.llnl.gov/computing_resources/sequoia/` (visited April 2013).

[13] J. R. Levine. *Linkers and Loaders*. October 1999.

[14] P. Miller. Parallel, Distributed Scripting with Python. In *Proceedings of the 3rd Linux Clusters Institute International Conference on Linux Cluster: The HPC Revolution*, Chatham, MA, USA, 2002. *IEEE* Computer Society Press, Los Alamitos, CA.

[15] J. A. Rathkopf, D. S. Miller, J. M. Owen, L. M. Stuart, M. R. Zika, P. G. Eltgroth, N. K. Madsen, K. P. McCandless, P. F. Nowak, M. K. Nemanic, N. A. Gentile, N. D. Keen, and T. S. Palmer. KULL: LLNL's ASCI Inertial Confinement Fusion Simulation Code. *Physor 2000, ANS Topical Meeting on Advances in Reactor Physics and Mathematics and Computation into the Next Millennium*, May 2000.

[16] rtld-audit man page. http://man7.org/linux/man-pages/man7/rtld-audit.7.html (visited April 2013).

[17] W. Yu, R. Noronha, S. Liang, and D. K. Panda. Benefits of High Speed Interconnects to Cluster File Systems: A Case Study with Lustre. In *The International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.