Zoid [Zepto OS IO Daemon] Design

Ivan Beschastnikh

Ofice of Science, Science Undergraduate Laboratory Internship(SULI) Program

University of Chicago

Argonne National Laboratory

Argonne Illinois

August 19, 2005

Prepared in partial fulfillment of the requirements of the Office of Science, US Department of Energy Science Undegraduate Laboratory Internship (SULI) Program under the direction of Dr. Peter Beckman in the Math and Computer Science Division at Argonne National Laboratory.

Participant: _____

Signature

Research Advisor: _____

Signature

Contents

Abstract

Zoid [Zepto OS IO Daemon]. IVAN BESCHASTNIKH (University of Chicago, Chicago, IL 60637) PETER BECKMAN (Argonne National Laboratory, Argonne, IL 60439).

The design of IBM's Blue Gene[1] family of super computers scales to thousands of nodes by offloading all system calls made by a compute node to a designated IO node. To facilitate and make the porting of software possible, the system calling interface is transparent to software written for POSIX compliant systems. The native mechanism responsible for servicing forwarded system calls is the Compute IO Daemon (CIOD). This paper introduces and describes the design of ZeptoOS IO Daemon (Zoid), an alternative to CIOD. Zoid is an attempt to construct a scalable system call forwarding mechanism that is designed to emulate CIOD on the BG/L, and integrate with ZeptoOS[2], a platform independent research operating system meant to scale to systems with millions of CPUs. Zoid has a flexible design which allows user defined behaviour per system call, varying consistency semantics and a choice of cache policies. Another critical feature of Zoid is that it is an open platform for research. Open Source lets scientists explore advanced functionality and new algorithms for BG/L that would otherwise not be possible. Most importantly, Zoid and the ZeptoOS suite is the only available open source choice for the BG/L architecture.

Introduction

Some recent innovations are pushing cluster technologies towards grid like settings by wiring commodity hardware into gigantic networks such as in [3]. These systems are used to achieve high levels of parallelization and aggregate throughput but not extreme computational power. Another approach is to minimize the size of the nodes resulting in lower energy consumption, heat output, and internode distance. This allows for very large number of CPUs with a low-latency fabric. The latter category is trying to break the petascale computing barrier with a variety of ingenious design decisions.

Many of these decisions were poured into IBM's Blue Gene/L (BG/L) architecture[4] which is at the time of this writing holds the number one position on the list of the fastest supercomputers in the world[5] with one of its models at Lawrence Livermore National Laboratory (LLNL).

One of the key features of BG/L is the Compute IO Daemon (CIOD), a system call forwarding mechanism responsible for relaying system calls from the compute nodes to dedicated IO nodes, servicing those system calls and returning the output back to the compute nodes. The IO nodes only service system calls for its group of compute nodes, at Argonne National Laboratory (ANL) the BG/L computer has a thirty two compute per one IO node ratio whereas LLNL has a denser configuration with eight compute per one IO node ratio.

CIOD which ships as part of BG/L is no more efficient than a general purpose call forwarder. CIOD is synchronous, single threaded, doesn't support any caching, and is not

configurable. To add to its lack of efficiency, CIOD is closed source, making any attempts at computer science research on the BG/L futile since CIOD provides a core service that cannot be easily isolated, replaced or removed.

In order to reap further benefits of the BG/L design, Zoid (Zepto OS IO Daemon) has been designed as an eventual replacement for CIOD. Zoid will be bundled with ZeptoOS, an open source software solution for BG/L and other architectures that may benefit from small, lightweight kernels.

The rest of this paper will focus on the important features of Zoid that make it a more suitable choice as a system call forwarding mechanism than CIOD such as configurability, caching, and a flexible consistency model. Details of Zoid's design and explanation of the innerworkings of the current implementation will also be discussed.

Overview of the design

Zoid server and client may have either a library implementation or an in kernel implementation. The current prototype is a library implementation, therefore most discussion will center around a user level server that runs as a daemon process and a client library which links with an application that has been modified to use system calls provided by the zoid library.

The server process is a multithreaded application which prethreads a fixed number of *serving threads* at startup. The *main thread* listens for incoming client connections and acts

as a scheduler between client system call requests and the serving threads which serve system call requests and spend the rest of their time sleeping.

The client and server communicate via the Zoid Data Protocol[6] over TCP, Myrinet or Infiniband. Besides the main channel of communication, the client and server maintain an out of band channel used for Zoid debugging information on the client side. This channel may also prove useful for boot and kernel message passing from a ZeptoOS compute node kernel in the future. The prototype only supports TCP transport.

Client

The Zoid client is a single threaded process which acts as a translator of all system calls made by another application into Zoid system call requests to the Zoid server. For the process to be completely transparent to the application, a kernel or glibc modification is required. The Zoid client prototype is a lightweight library that exports most of the file IO system calls supported by the linux kernel.

A challenging feature of the client side implementation for BG/L is that there is no polling mechanisms in the client code since polling system calls such as select and poll are not supported natively by the BG/L compute node kernel nor by the CIOD. This means that the Zoid client must be as synchronous as possible as asynchronous messages or requests introduce inordinate complexity.

The Zoid client may carry out complex functionalities other than merely forwarding and receiving syscall requests. From a prototype user perspective, Zoid system calls have a 'zoid_'

prefix to qualify them as system calls forwarded by zoid. Internally however, the Zoid client may process system call locally (eg. the umask syscall is a good syscall example that has been implemented only on the client side).

Server

The Zoid server has gone through a couple of designs. The recent multithreaded server has superceded the earlier multiprocess design which was appealing because multiple processes could deal with all data mutually exclusively, making file descriptor management easy and removing the need for mutexes. Upon further investigation, we have decided that it would be much easier to coordinate communication between threads rather than processes. A multi threaded design would ease implementation and would also benefit in speed due to caching. In a multithreaded server data is closely shared amongst the threads, so any client can be serviced by any one of the threads. This eventually leads to more advanced caching designs that speed up syscall processing.

The 2.4 Linux kernel schedules pthreads as regular processes but they can take advantage of caching as they have access to each other's pages. The 2.6 Kernel supports pthreads as lighterweight entities rather than full processes, making threads more advantageous as they have less system overhead.

At the moment the Zoid prototype supports the following system calls: time, gettimeofday, open, read, write, close, fstat, lseek, umask, printf, perror, strerror.

Scheduler

The server scheduler is responsible for assigning clients with outstanding system calls to servers. The prototype supports round robin scheduling, but there are many extensions that are being planned.

One important modification is to allow for inlined system calls. This would be very beneficial to two types of system calls. Those system calls that can be processed faster than the scheduling time, which depends on the scheduling algorithm, plus a context switch to the chosen serving thread. Also, those system calls that can be served right out of the server cache might be processed faster (in aggregate) if they are inlined with the main server thread than if they are scheduled.

Another important modification to the scheduler is to prioritize system calls based on a variety of criterea. For example, a system call that will cause a cache hit should almost always be scheduled ahead of any other outstanding system calls.

Caching

Caching is a core idea to Zoid's design. Because parallel applications tend to access similiar resources, a server side cache will diminish the impact on the resource and can alleviate scheduling overhead. Caching is also crucial for collective operations that explicitly trigger cache optimization.

Zoid opens one file per filename, and maintains a cache for those files that have a reference

count of two or greater. After file access, the memory allocated to a thread is hot and can service file IO operations to the same file with an offset within the range of the previous call much faster than that of any other server thread that have a cold cache. Therefore the caching policy must work closely with the scheduler to ensure maximum benefit from the local memory system cache.

Besides server side caching, the client can also maintain a cache. For example, if a file is opened by only one node for writing, the client may decide to use a write-back policy in which unless the server specifies that another node opened the file or until the file is closed or sync'ed to disk, the client maintains a local cache of the changes to be eventually sent to the server as a combination of write and lseek system calls.

One other idea is to be able to maintain client caches from the server side. If a server knows the access pattern of the nodes (eg. collective read operation on all the nodes) then it can short circuit future node read requests by *prefetching* results to clients and warming the caches for later access.

These kind of nontrivial behaviours calls for a set of complex consistency models that must ensure consistency for all IO access patterns.

Consistency Models

The POSIX standard has a serial consistency model. This model is easy to implement but requires locks to sequentially order certain operations. On multi processor and clusters

machines locks can become a bottleneck. PVFS (Parallel Virtual File System)[7] is an example of a system that is extremely successfull because of its relaxed consistency model that avoids control lock services. Zoid likewise would benefit tremendously from a relaxed consistency model that allows for resource modifications to propogate in parallel.

The current prototype implements the serial consistency model. We plan however to integrate a variety of models that can be selected during server and client startup. As an example, consider a client that opens a file and writes to it. If the file hasn't been opened by other nodes, the file consistency can be maintain on the client and synchronyzed only upon a close or a sync syscall. This saves round trip times and boosts efficiency. The behaviour must change however if the file is then opened by multiple nodes.

Many of these and similiar design decisions have been already addressed and implemented in PVFS(2) but the consistency models for Zoid will go beyond file IO, and will support socket IO, shared memory, and other IO mechanisms.

Security

Zoid is meant to be run on dedicated cluster machines that usually have a specially allocated subnet for their operations and are protected from potentially malicious hosts. Because of this Zoid uses host-based security and assumes that all hosts in the system are friendly and follow the Zoid Data Protocol correctly.

Future Work

The next immediate step in Zoid development is extensive benchmarking of the prototype to study file IO speeds. Based on the benchmarking results, the networking code may need streamlining. One difficulty is that the Zoid prototype cannot be compared directly to the CIOD. In order to make a valid comparison, Zoid must be running under BG/L, a feat requiring a ZeptoOS compute node kernel. This is still in the works. Therefore current benchmarks are being done on x86 linux clusters.

Zoid's benefits can be far reaching and its functionality doesn't have to be limited to forwarding system calls. As an example, one use of the state stored by the Zoid server for all of the compute nodes is the ability to do check point restart. If a compute node crashes, it's memory can be stored and the application can be restarted on another node at the exact point the crash happened. Applications can be paused, saved, swapped, etc. Zoid can foster in a whole new level of job management and redefine parallel application debugging.

Another important research area is that of collective operations. Parallel applications usually hammer at IO resources due to poorly parallelized code. One solution is for Zoid to support collective operations that the application programmer can use to optimize application performance. The same holds true for integration of Zoid with PVFS. By supporting PVFS, Zoid will have gained lots of ground over CIOD which as of now does not support PVFS, the most ubiquitous file system used on super computers.

One of the inherent design problems with Zoid is that the prototype loses functionality and performance due to a library style of implementation. Our future work's aim is to design a Linux kernel module that will integrate with the 2.6 Linux kernel to eliminate at least one copy, and two user level to kernel level transitions.

Related Work

Because Zoid involves so many areas of computer science research, there are many previously researched and ongoing research areas that can benefit Zoid's ongoing design and implementation. The closest works have been done by the OpenMosix[8] and OpenSSI[9] projects although their systems do not share the ZeptoOS goal of lightweight kernels.

The ZeptoOS light weight kernels which Zoid will eventually inhabit have been studied before. Particularly, their usefullness has been documented in [10]. Other designs such as the more popular catamount[11] implementation are also available.

There is lots of work on caching algorithms widely used in web servers[12], kernel scheduler design, NFS server design, and other software. Scheduling has also been closely studied ever since operating systems became multiprocessed and the internet came into being.

Acknowledgements

This research was conducted at Argonne National Laboratory. I would like to express my most sincere gratitude to my mentor Pete Beckman for fueling me throughout the summer with challenging ideas and failsafe methods on all our projects. All ZeptoOS members at

ANL took part in designing parts of Zoid either by direct involvement or by providing extremely useful comments and feedback. I especially would like to thank Kazutomo Yoshii for technical discussions, Cameron Cooper for work on the test cases and Chiba benchmarks, Nick Trebon for benchmarking parts of the code, Kamil Iskra for useful tips, and Yitzhak Wasileski for good company. I also want to thank the Department of Energy, the Office of Science, and the SULI Program for granting me the opportunity to participate in the program.

References

[1] A. Gara, et al. "Overview of the Blue Gene/L system architecture." in IBM Journal of Research and Development, Vol. 49, 2005, pp. 195-212.

[2] "ZeptoOS Project." http://www.mcs.anl.gov/zeptoos

[3] S. Ghemawat, H. Gobioff, S.T. Leung, "The Google File System" in Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003, pp. 29-43

[4] J. E. Moreira, et al. "Blue Gene/L programming and operating environment." in IBM Journal of Research and Development, Vol. 49, 2005, pp. 367-376

[5] "TOP500 List 06/2005.", http://www.top500.org/lists/plists.php?Y=2005&M=06

[6] "Zoid Data Protocol." I. Beschastnikh, http://www.mcs.anl.gov/zeptoos/docs/papers/zoid_protocol

[7] "PVFS (Parallel Virtual File System) Project.", http://www.pvfs.org/pvfs2/

[8] "OpenMosix Project.", http://openmosix.sourceforge.net

[9] "OpenSSI Project.", http://openssi.org/cgi-bin/view?page=openssi.html

[10] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, pp. 55-72.

[11] S. Kelly and R. Brightwell, "Software Architecture of the Light Weight Kernel, Catamount," in Proceedings of the 2005 Cray User Group Conference, pp. 12-23.

[12] E. Cohen, B. Krishnamurthy, J. Rexford, "Efficient Algorithms for Predicting Requests to Web Servers" in INFOCOM 1999, pp. 284-293.