Block Mason

Dutch T. Meyer[*†], *Brendan Cully*[†], *Jake Wires*[‡],
Norman C. Hutchinson[†] *and Andrew Warfield*[† ‡]
[†] *Department of Computer Science*
University of British Columbia
[‡] *Citrix, Inc*

Abstract

Hardware virtualization gives administrators the flexibility to rapidly create, destroy and relocate virtual machines across physical hosts. Unfortunately, the storage systems upon which these systems depend are not nearly as agile. To facilitate the rapid, safe development of block devices that can meet the needs of virtual machines, we present the Block Mason virtual block device framework. Although the block device interface is simple and intuitive, block devices themselves must generally be implemented in the operating system kernel, an environment which is neither simple nor portable. Block Mason allows users to build small, reusable block processing elements in user space, and to connect them together into powerful composite modules using a simple declarative graph language. Although the environment emphasizes simplicity for developers and end users, it includes built-in support for powerful operations like live reconfiguration and dependency tracking.

1 Introduction

Beneath the simplicity of the block device interface lies a deep and varied set of techniques to store and retrieve blocks. Requests may be routed, filtered or duplicated, encrypted, compressed, or checksummed, to name just a few possible transformations. From these basic operations, complex higher-level services such as incremental backup, content-based addressing, copy-on-write, mirroring, migration and distributed storage services can be constructed. With knowledge of the semantics of I/O request streams, [1, 10] this list grows further to include capabilities traditionally reserved for file systems [9]. But in spite of the significant benefits and wide applicability of such services, they have not been broadly adopted.

For developers, the attractive simplicity of the block interface is obscured by the difficulty of kernel programming. Kernel extension writing is dangerous and offers deliberately limited functionality. Storage providers are obliged to frequently reinvent their own libraries for configuration and management, costing developer time and resulting in inconsistent interfaces for end users. These restrictions are becoming increasingly onerous due to the complex, dynamic, and unique demands placed on storage by today's highly networked, virtualized environments. Despite these challenges, the block interface remains the natural place to provide storage to virtual environments, because it is simple and highly portable across virtual machines.

Block Mason offers developers a powerful and easy-to-use environment for developing reusable storage elements, and lets administrators combine these elements to easily create powerful storage facilities. Its approach, conceptually similar to the Click [4] modular router, is to treat storage devices as a dynamically reconfigurable graph of simple block request handlers. Modules may be written in high-level languages, and encapsulate simple routing, analysis or transformation operations. Our VM-based architecture offers an extensible set of storage operations, and a convenient platform for user-mode development with cross-platform compatibility.

[*]Written while on internship with Citrix, Inc.

2 Architecture

Block Mason is a redesign of the blktap [11] interface, which exports block requests from a guest VM to user-mode in a privileged domain. We borrow terminology from Click [4], as our system is conceptually similar.[1]

Block Mason provides two primary interfaces: one for module authors, and another for users–typically system administrators–who assemble modules into new storage designs. We will consider our architecture primarily from the perspective of the latter in order to be most illustrative. In addition to what is presented below, services common across elements (e.g. error reporting, request forwarding, etc.) are provided to modules as part of the Block Mason API.

2.1 Elements

Elements are re-usable modules that perform data processing, routing, and analysis. Our intent is that system designers and maintainers will have a large pool of pre-made elements available. This code re-use will enable development to be quick and safe. In addition, our framework facilitates the design of new elements for special purpose tasks.

Each element is configured with a type and any element-specific parameters. Table 1 shows a sample from one of our configuration files: the `debug_log` element passes requests through unmodified, recording them in a log.

```
element MyDebug
  (type debug_log)
  (log_file /var/log/request_log)
```

Table 1: Element configuration syntax.

A significant consideration in our design is ensuring that elements are easy to write. We expect that most module authors will work in high-level languages. Still, our current linearization module (which aggregates any number of block devices or files into a single volume) is implemented as 26 lines of C and 73 lines of template code.

[1] Important differences do arise from our focus on storage as opposed to networking. Request forwarding is done on two-way request/response channels in Block Mason, and our elements tend to be relatively coarse grained operations, acting on whole-blocks.

2.2 Ports

Passing requests between elements is done through *ports*. Ports are created in conjunction with elements, and are given meaningful names to identify their function. Administrators connect elements together by matching the input port on one element to the output port on another. An example of our syntax is shown in Table 2; here an edge connects the `debug_log` element to a block device.

```
(MyDebug, out) -> (LocalDisk, in)
```

Table 2: Edge configuration syntax.

Elements can route requests to any of their output ports, and can register to receive success or error notices when the request completes.

Our scheduler takes each request from the outgoing ports and re-queues it on the appropriate incoming port of the next element. Graphs in Block Mason are event-driven, with elements operating asynchrononously. At any given time, many requests may be in flight, as batching is critical to achieving good performance. Dependency tracking is provided in the API for modules, such that authors can ensure disk consistency by ordering writes. Our dependency tracking mechanism is borrowed from Parallax [6].

Currently elements share an address space, but since block requests are serializable, it is straightforward to support ports that cross process, VM, or network boundaries. This facilitates the use of domain specific languages, recovery from failed elements, and the creation of more complicated network services, similar in scope to Petal [5].

2.3 Live Updates

The graph of a running device can be modified on the fly in order to add, remove, or reconfigure features. This allows designers to adopt flexible storage policies. As requirements change over time, the storage system can evolve accordingly.

Our command-line tool allows an administrator to display and manipulate the graph, then reload the current graph with any element or edge modifications. To accomplish this, the stream of requests is temporarily

paused at the source, and outstanding events are quiesced. The graph is then destroyed and rebuilt as if it were being constructed for the first time. Users of the device will experience a brief period of increased request latency but remain otherwise unaffected. This ability is very useful in performing run-time reconfigurations, like checkpointing a chained image file, or adding (and later removing) probe modules for profiling the block request stream.

2.4 Dependency Tracking

To facilitate the construction of more complicated services our framework provides support for the tracking of inter-request dependencies. This can be used to ensure ordering between requests and is useful for providing data consistency or persistent logging guarantees. Since this feature is provided by Block Mason, any module can make use of the capability. Our intention is that features representing graph-level concerns, such as this one, can be moved into the architecture to ease the burden on module developers.

Modules may order any number of requests by assigning dependencies between them. These dependencies dictate that the dependent request not be issued to persistent storage before its associated independent request. To introduce a dependency, a module author simply uses the `td_add_dependency(dep_req, indep_req)` call. This tags both `dep_req` and `indep_req` requests, such that they will be tracked in our scheduler. New modules that wish to track dependencies can also check the state of any request with the `td_is_independent(req)` call.

3 Case Studies

To demonstrate the utility of Block Mason, we now briefly discuss two example block-level services that have been constructed with it. In addition to these examples, we are in the process of porting Parallax [6] from its current blktap-based implementation to run as a collection of Block Mason components.

3.1 Live Volume Migration

To demonstrate the application of our architecture, we show how a maintenance task can be accomplished using a small number of general-purpose modules. While small VM-based servers may begin operation on individual physical hosts using local disks, many deployments will evolve toward the use of network-based storage such as NFS or iSCSI. Unfortunately, migrating VM images from one storage device to another involves large amounts of bulk data transfer and may incur considerable downtime. In this case, Block Mason can be configured to provide *live* disk migration, as shown in Figure 1, where the in-use image is moved from one storage target to another under the feet of the running VM.

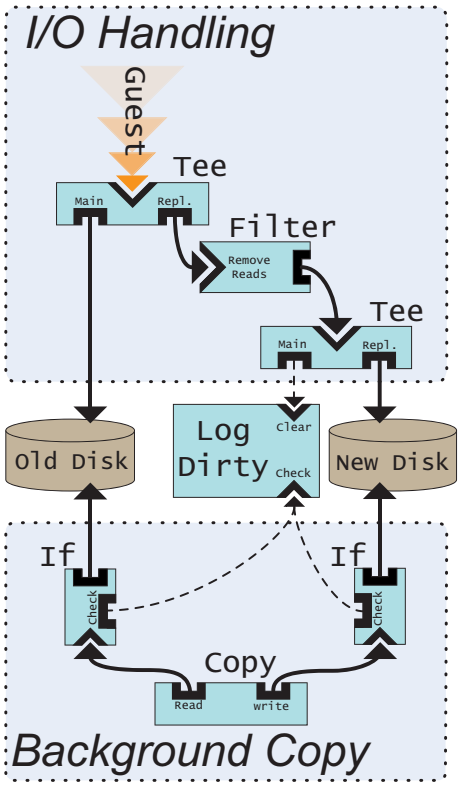

Figure 1: Live migration configuration.

3.1.1 Service Construction

The system is constructed from five distinct reusable elements, and is designed to allow the stream of disk I/O from the guest system to continue while copying blocks from the original disk in the background. Each distinct element is explained below:

- **Tee** – Duplicate an incoming request stream between any number of output ports. The guest receives completions for I/O from the original disk only.

- **Filter** – Filters requests according to some criteria. In Figure 1 read requests are filtered.

- **Log Dirty** – This filter module maintains a list of block addresses which have not yet been synchronized between two disks. In Figure 1, all blocks are initially set to dirty. As blocks are written to the target disk, their addresses are cleared from the filter.

- **If** – Tests the request address against a filter module, only forwarding requests corresponding to addresses present in the filter. By connecting this to the log dirty module, we drop non-dirtied blocks.

- **Copy** – The copy module is in charge of driving the background copy operation with read and write requests. It is initialized to walk through the entire range of the original disk.

3.1.2 Operation

This configuration is comprised of two distinct operations. The I/O handling operation mirrors requests across both disks, while keeping the log dirty module up to date. At the same time, a background copy operation migrates data from the old disk. In production, an error handler should be placed between the tee and the new disk to catch any failures on that device. The handler can then abort the migration or retry the request.

Our system handles the two potential races between the background copy and the I/O handling. First, the If module on the rightmost path ensures that new writes are not accidently overwritten.[2] Second, Block Mason handles races involving conflicting writes issued concurrently at end points, by delaying the latter request.

Once all blocks are marked clean in the log dirty module, the graph can be replaced with direct access to the disk. It is also possible to disable Block Mason's user-mode framework and issue future requests directly to the block device.

[2]The If module on the leftmost path is a performance optimization.

Many further improvements to this model are possible: quality of service modules could prioritize traffic from the guest, notification modules could identify when the process is complete and email the administrator to that effect.

3.2 Cloud Storage

Cloud storage is an attractive service, but access to it requires internet protocols, such as HTTP and REST, that differ greatly from the block interface used for most other storage systems. It is therefore unsurprising that the usage of cloud storage is heavily weighted towards web-based applications [2]. Yet the location independence and high availability offered by such services is generally desirable. The following module demonstrates how an Amazon S3 disk can be made available as a block device to any virtual machine.

S3 clients operate on *buckets* which are containers for key/value pairs, and fees are assessed for data transfer. Our S3 module treats buckets as volumes, with block addresses serving as keys. It acts as a request sink, marshalling them into REST commands conforming to Amazon's API which it forwards to an S3 server.

This allows us to use a cloud storage service as a block device. An example is shown in Figure 2. Here we are using our module to host a guest VM's file system from the cloud. We have added a local cache to provide fast access to a working set's worth of data (and to reduce access fees), and encrypt all data prior to network transmission. Other modes of operation, such as secure offline backup, are also possible.

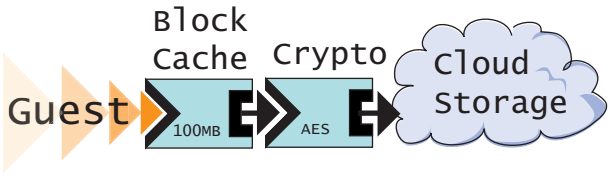

Figure 2: A cloud-based block device created with Block Mason modules.

Performance-sensitive users will likely want to extend this design with other modules. Communication overheads make 4k pages inefficient, but with Block Mason it is easy to create a module that adjusts block sizes. Semantically intelligent modules could preferen-

tially divert valuable data into the cloud. Alternatively, with a measurement module that was aware of the pricing structure, spending limitations could be enforced, even across a cluster of VMs.

4 Related Work

In addition to previously discussed work, our goals are very much in line with those of FiST [12], which is effectively a file system compiler that supports many platforms. Our efforts instead focus on a cross-platform block-level architecture. Similarly, all previous work on stackable file systems [3] seeks to ease development with a more modular architecture. Our approach is an extension of these prior efforts, albeit with a more restricted focus.

Modules in Block Mason can operate entirely in user-mode and may reroute requests dynamically. The flexible nature with which they can be composed allows building complex systems from simple reusable elements. Configuration is also simplified through the use of a human-readable, declarative syntax. These features differentiate our system from the Linux device mapper, in which all data processing must occur within kernel modules and configuration is limited by the *ioctl* interface.

5 Conclusion

Our examples of a live migration service and cloud storage volume demonstrate the versatility and usefulness of Block Mason. Our system can be reconfigured underneath a running guest VM, using a declarative configuration language. This allows administrators to perform dynamic maintenance and refactoring operations on their storage systems without service interruptions.

By building common routing, analysis, and modification operations into simple modules, we also show how the process of creating new services at the block level can be made easier. Since devices are composed as a graph of elements, Block Mason can provide very complicated features using mature, well-tested modules. Our system handles much of the complexity and subtlety of request routing and dependency tracking interactions, freeing designers to focus on the unique concerns of their services.

Our prototype has a functional scheduler and syntax parser. We can create and run complex devices, including elements with arbitrary port configurations. Edges and elements can be added to or removed from a running system without interruption. Dependency relationships can be assigned and are correctly tracked across the graph. We are in the process of building a larger and more dynamic set of modules, creating a stronger definition for the synthetic request channels and improving error handling. Additionally, this work inspires several relevant research directions.

One challenge is ensuring that there are sufficient safeguards in place to avoid data corruption due to a poorly designed configuration. Mounting a volume with the wrong elements in place, or in the wrong orientation could easily result in data loss. Similarly, if a module itself is made unavailable, an entire volume may be rendered unreadable. One approach to this problem would be to sign each volume (or even block) with the module configuration used to create it. Alternatively, Block Mason makes it feasible to wrap complicated operations with a copy-on-write module, so the underlying data is protected.

Block Mason exposes a very powerful interface, enabling the creation of very complicated storage systems. By reimplementing an existing block-level system, e.g. [7], we can provide a basis for measuring our system's performance and expressiveness. We hope that it will also show that future block-based systems can be built more easily through module reuse.

Similarly, Block Mason may be able to incorporate traditionally higher-level file system mechanisms into the block layer. Linux's request elevator, for example, could be reimplemented as a set of elements, which would allow users more control over batching and scheduling policies.

When ports are made to cross protection boundaries, high-level domain-specific languages can incrementally replace our current modules. It is likely that a language written specifically for routing and manipulating block requests could be made far simpler than C. Such a language may also allow static and run-time analysis to establish correctness, perhaps by incorporating ideas from [8]. Similarly, our configuration language could be extended to support declarative invariants on operation, such as *"all data is encrypted before it reaches disk."*

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 43–56, New York, NY, USA, 2001. ACM.

[2] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264, New York, NY, USA, 2008. ACM.

[3] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Technical report, 1991.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[5] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.

[6] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 41–54, New York, NY, USA, 2008. ACM.

[7] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX Conference on File and Storage Technologies*, Monterey,CA, 2002.

[8] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[9] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with d-graid. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 15–30, Berkeley, CA, USA, 2004. USENIX Association.

[10] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[11] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 2005. USENIX Association.

[12] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *In Proceedings of the Annual USENIX Technical Conference*, pages 55–70. USENIX Association, 2000.