MultiBlockD

An Architecture for Flexible Storage Device Delegation

Summary:
MultiBlockD will provide a mechanism for the specific details of how requests to block devices are processed to be controlled by a user-space program. It aims to be a generalization of things like Windows ReadyBoost and ZFS L2ARC, which use solid-state disk to cache/buffer/extend spinning disks. It will consist of two parts, a kernel module with a virtual block device that delegates to slaves, and a user-space program running as root that will determine mappings between the virtual and real devices. The main artifact will be a general block-device-combiner that can securely take mappings from user-space.

Oversight is being provided by Professors Gene Cooperman and Peter Desnoyers of Northeastern University.

Motivation:
Basic caching algorithms and software RAID policies should be trivial to implement with little overhead. However, in future work, this can be extended to a system with more complex heuristics. This can allow users to take better advantage of flash-based disks in the storage hierarchy, which is relevant to Professor Desnoyers' research. Also, the user-space program could be extended to allow dynamic mappings based on non-trivial external communication. This could be used for rapid development of new load-balancing and reliability architectures.

Plan:
We have strict time constraints, so progress must be made quickly. Further refinement can be made in later revisions.

The project will be split into 3 parts:

1. Kernel module to deal with block devices and request handling (headed by Matthew Strax-Haber)
2. User-space program to determine the destination and mapping of user requests (headed by Matt Ivester)
3. Mechanism for dealing with communication between the two (headed by Jed Davis)

Our primary goal is to get a device driver working and implement something resembling a direct-mapped caching policy, with a primary disk and a cache device. The policy language, as well as more novel uses of this architecture, are secondary, but will be considered when implementing APIs.

Related information on Confluence Wiki:
Design Notes (work in progress)
Matt's Development Environment (for reference)
Reference Materials
Pseudo-Code for Multi-Block (outdated – we're not using this architecture anymore)

Note:
All rights reserved, Matthew Strax-Haber, Matt Ivester, and Jedidiah Davis
Labels
multiblock, computer, architecture, research, raid, caching, virtualization, flash, l2arc, readyboost, block, device, linux

Children (5)
Design Notes

The General View

When the kernel gets an I/O request for a synthetic block device, it tells the policy process the location and size and read/write flag; the policy process responds with instructions for real I/O operations to perform to satisfy the request.

Somewhat More Specific

The request may cover multiple “cache lines” or “stripes” or otherwise need to be fragmented to deal with. So, the response will take the form of a collection of segments, referencing an extent of the request and specifying how to deal with that part of it.

Processing each such segment might involve a simple read or write from/to one disk, or it may be more complicated — if a region larger than the segment is being entered into a cache at the same time as it is being overwritten, for example, a read-modify-write sequence is needed, as well as buffer space separate from that of the original request. Thus, each segment is handled by a sequence of operations, each of which is a read, a write, or a copy to/from the main buffer.

This could also be approached more generally, by having the policy express a general DAG of operations (along the lines of the architecture used internally by RAIDframe, perhaps) instead of this more restricted form. However, we conjecture that this added structure will simplify the implementation without costing expressiveness.

A Word on Concurrency

It is important that we not violate the expected block device semantics by, for example, reordering two writes to the same location. Given that restriction, we would like to impose as little unnecessary serialization as possible for the sake of performance, but we have to be careful, because the I/O operations we get may expand into sequences of multiple I/Os on the underlying disks, leading to situations like this:

```
A =========       A1 =========
B ========= --->  B1 =========
         B2 ========= X
         A2 ========= X
```

Clearly the sequence of commands within a segment must in general be processed in the order given. On the other hand, it seems reasonable to require that the segments making up the response to a given request not have ordering dependencies between them; *i.e.*, to allow them to be run in parallel with each other.

This leaves the question of ordering between different requests. One approach is to require that the results be consistent with serial execution, and require that the kernel side figure out what it can safely parallelize. This seems to impose more complication on the kernel side than is desirable. Another approach is to require the userspace policy to track which physical I/Os might be in-flight and hold back responses or otherwise express ordering requirements as necessary. This is likely to complicate the interface, and perhaps introduce inefficiency due to the policy's separation.
from the actual I/O.

The compromise we have adopted, at least for the time being, is for each segment to have a serialization key, such that the policy guarantees that two segments with different serialization keys can be reordered. (For example, for a simple RAM-cache-like approach, the cache line number would suffice.) The motivation is that the kernel side can serialize only within each serialization key, and hopefully this will be good enough without being especially complicated.

**Copyin, Copyout**

There are two issues of note with the user/kernel interface: we'd like to have as few context switches as feasible, and the available primitives generally take the form of a call from user space into kernel space. This latter seems as if it might be a problem for the "kernel … tells the policy process" above, but there is a standard solution in the form of a kind of inversion of control: the user process makes a system call which blocks until the kernel has something to return.

Concretely, we will have some kind of system entry, probably an ioctl on the pseudo-disk device, carrying a pointer to one of these:

```c
struct mblk_stuff
{
    struct mblk_request *requests; /* kernel -> user */
    size_t nrequests;
    struct mblk_segment *segments; /* user -> kernel */
    size_t nsegments;
    struct mblk_command *commands; /* user -> kernel */
    size_t ncommands;
};
```

The effect of the call is to both submit the segments and commands given by their corresponding pointer/length pairs and block until new requests are available, at which point they are copied into the array given by requests; nrequests is the capacity of that array on entry and the number of elements used on return.

Note in particular that the segments/commands are performed asynchronously; once all of the commands for a request complete, the original I/O is returned. This interface could be straightforwardly extended to notify the policy of real I/O completion (and, in particular, to give timing information it could use to dynamically adjust its behavior), as well as to report errors and allow a fallback strategy to be submitted in response, but neither of these has been done here.

**More Details**

(...)

(to be continued...)
Reference Materials

Added by Matthew Strax-Haber, last edited by Jedidiah Davis on Dec 10, 2009

Links/Resources

- http://www.linuxjournal.com/article/2890
- http://users.evtek.fi/~tk/rtos/writing_linux_device_d.html
- http://en.wikipedia.org/wiki/Berkeley_Packet_Filter — initially we were planning a BPF-like approach to expressing the cache control program, but later decided against it.

Relevant files in Linux source

- drivers/block/loop.c — should be able to crib driver stuff from here
- include/linux/majors.h — we'll need a major number (or allocate one?)
- (some Makefile somewhere)