Design Commentary

Added by Jedidiah Davis, last edited by Jedidiah Davis on Dec 21, 2009

The Goal

New secondary storage technologies, such as NAND Flash, occupy a position in the storage hierarchy intermediate between RAM and conventional disks, but with characteristics different from either. Some interest in these technologies is, therefore, in using them as an intermediate level of caching; examples include Sun's L2ARC and Microsoft's ReadyBoost. It would be desirable to have a platform for experimentation with such techniques, as well as for more general forms of combining multiple block devices into one.

The resulting artifact would be a pseudo-disk-driver similar to those used for software RAID or block-level encryption, but with different goals.

How We Are Not Doing This

There is existing work in the form of dm-cache, a backend module for the Linux device mapper, which uses one disk as a cache for another — according to a hard-coded policy with only the most limited tunability. Obviously we intend to allow more flexibility than that.

A more general approach would be a block device driver (or device mapper backend) with a plug-in interface by which loadable modules can register themselves as policies. This is a common pattern, used by the device mapper itself for extensions such as dm-cache, the packet filtering framework iptables, device drivers in general, and so on. This has the advantages of allowing any policy that can be expressed in C code, and can draw upon any information available through kernel internal APIs. But it also has disadvantages: all of the programming restriction of the kernel environment apply; any bugs can take down the entire system, or even damage unrelated data; and policies will be tied to kernel interfaces that are not even remotely portable and may be broken even by minor upgrades within a single system.

Another approach, which we spent some time considering, is to have the policy be a program for a simple virtual machine interpreted by the kernel — along the lines of the Berkeley Packet Filter, where packet capture utilities can supply filter programs to be run in kernel space to avoid the overhead of copying unwanted traffic to userspace. This also has several advantages: the policies can be written in a high-level domain specific language tailored for the problem space; the interpreter can easily be made safe, thus isolating the kernel from faults; the language could potentially even have statics that rule out certain types of logical errors; and this nicely enforces a layer of abstraction from system-specific details. On the other hand, keeping the system safe from misbehaving policy programs (including runaway resource usage) will tend to sharply curtail the kind of algorithms that can be expressed, efficiently or even at all; any sources of information must be explicitly connected to the virtual machine; and, perhaps most importantly, this would have resulted in a much more ambitious project than the time available would allow for.

The General View

To allow the maximum amount of flexibility in terms of caching policy, we have chosen to have decisions made by a userspace policy server; this allows for rapid development of policy code, a certain level of fault isolation, the full expressiveness of any programming language that can interoperate with C, and the use of any OS functionality (e.g.,
hardware monitoring information) that has a userspace API. This does however incur overhead due to context switching and scheduling latency, and while designing the user/kernel interface is much easier than designing a language, it nonetheless requires some thought.

In particular, 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. The following sections go into more detail about this interface.

**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 -------
   ------ X
A 2 ------ 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 (through both the introduction of additional context switches and the need to sometimes wait for I/O completion before continuing.)

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 will serialize only within each serialization key. In simple cases, this seems to allow for optimal or nearly optimal parallelization, although we did not formally analyze this mathematically.

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 (analogously to calls like recvfrom).

Note in particular that the segments/commands are performed asynchronously; once all of the commands for a request complete, the original I/O is completed. This interface could be straightforwardly extended to notify the policy of real I/O completion. This is necessary for some caching or logging schemes (such as certain types of journaling), and would allow for performance analysis of the underlying devices (which might also influence the caching policy). Extending this further with ability to report errors and request an alternative execution plan would allow for more types of policies to be expressed (e.g. RAID mirroring or multipath access with failover), as well as the ability to continue functioning if, say, the USB key being used as your cache is suddenly removed from the system.

More Details

The current header file for the C API interface to the kernel: mblk.h

See: Userspace policy prototyping for more discussion of how this API is used to process I/O.

See also The Old Interface and its attached files for the first draft of the interface, some simple (but untested) policy code written against it, and some brief discussion of its rationale.

There is also a brief experience report on an attempt at User-Level Prototyping of the Kernel Driver, which did not make nearly as much progress as hoped due to the technology involved being more temperamental than anticipated, as well as time limitations.
Children (1)

The Old Interface