THE HAMMER FILESYSTEM

Matthew Dillon, 21 June 2008

Section 1 – Overview

Hammer is a new BSD file-system written by the author. Design on the file-system began in early 2007 and took approximately 9 months, and the implementation phase following took another 9 months. Hammer's first exposure to the FOSS community will be as part of the DragonFly 2.0 release in July 2008.

It is my intention to make Hammer the next-generation file-system for BSD systems and I will support any BSD or Linux port as fully as I can. Linux of course already has a number of very good file-systems, though none of them (that I know of) go quite as far as I am trying to go.

From inception I laid down a number of requirements for what Hammer absolutely had to be capable of achieving:

- Instant or nearly-instant mount. No fsck needed on crash recovery.
- Full history retention with easy access, via the live file-system. That means being able to CD into snapshots of the file-system and being able to access previous versions of a file (or directory tree), whether they have been deleted or not.
- Good performance.
- Queue-less, incremental mirroring.
- Ability to re-optimize the layout in the background, on the live file-system.
- No i-node limitations.
- Extensible media structures to reduce compatibility issues as time progresses.
- Support for very large file-systems (up to 1 exabyte)
- Data integrity checks, including high-level snapshot tar md5s.
- Not hamstring a clustered implementation, which has its own list of requirements.
- Not hamstring a remote access implementation.

During the design phase of the file-system numerous storage models were tested with an eye towards crash recovery and history retention. In particular, I tried to compartmentalize updates by chopping the file-system up into clusters in order to improve parallelism. I also considered a propagate-to-root block pointer system for a short while but it simply was not compatible with the history retention requirement. One thing that became abundantly clear to me as I transitioned from design to implementation was that my attempt to avoid laying down UNDO information created massive complications in the implementation. For example, physical clustering of updates still required code to deal with the case where an atomic update could not be made entirely within a cluster. My other attempts equally hamstrung the implementation and I eventually abandoned the cluster model.

Low level storage management presented similar issues. I had originally planned a fine-grained low level storage allocation model for the file-system but it became clear very early on in the implementation phase that the complexity of a fine-grained allocation model was antithetical to the idea of retaining the full file-system history by default. What I eventually came up with is described in this section, and has turned out to be a major source of Hammer's performance.

Other portions of the original design survived, particularly the use of a B-Tree to index everything, though the radix was increased late in the implementation phase from 16 to 64. Given this brief history of the more troublesome aspects of the design, here is a list of Hammer's features as of its first release:

- UNDO records for meta-data only. Data is not (never!) immediately overwritten so no UNDO is needed for file data. All meta-data operations such as i-node and B-Tree updates, as well as the operation of the low level storage allocator, generate UNDO records. The UNDO records are stored in a fixed-sized tail-chasing FIFO with a start and end index stored in the volume header.

  If a crash occurs, Hammer automatically runs all open UNDO records to bring the file-system back into a fully coherent state. This usually does not take more then a few seconds regardless of the size of the file-system. If the file-system is mounted read-only the UNDO records are still run, but the related buffers are locked into memory and not written back to disk until/unless the file-system is remounted read-write.

- I-nodes, directory entries, and file data are all indexed with a single global B-Tree. Each B-Tree element has a create transaction id and a delete transaction id. History is formed when deleting or replacing an element simply by filling in its delete transaction id and then inserting a new element. Snapshot access is governed by an as-of transaction id which filters all accesses to the B-Tree, making visible only those records (many of which might have been marked deleted after the as-of id) and thus resulting in a snapshot of the file-system as of some point in the past.

  This use of the B-Tree added considerable complexity with regards to how lookups were made. Because I wanted the as-of capability to be native to the B-Tree's operation, and because searches on delete transaction ids are inequalities rather then exact matches, the B-Tree code had to deal with numerous special cases. To improve performance the B-Tree (actually a B+Tree) was modified to give the internal nodes one less element and provide a right-bounding element to allow Hammer to cache random pointers into the B-Tree as a starting-point for searches, without having to retrace its steps.

  This means that making a snapshot of a Hammer file-system requires nothing more then creating a single soft-link with a transaction-id extension, pointing to the root of the file-system. Backups made easy. Hammer provides a utility which allows old data that you really do want to delete to be destroyed, freeing up space on the media. A standard rm operation does not actually destroy any information unless the file-system is mounted explicitly with the “nohistory” flag or the directory hierarchy is flagged “nohistory” with the chflags program.

- Performance is very good. Extremely good for a first release, in fact. Front-end operations are completely decoupled from back-end operations, including for rename and truncation, and front-end bulk writes are able to reserve space on the media without modifying the storage allocation layer and thus issue bulk writes directly to the media from the front-end. The back-end then deals with updating the B-Tree after the fact.
- As of this writing it is not known whether mirroring will be available in the first release. The B-Tree element structure has a serial number field which is intended to propagate upwards in the B-Tree when mirroring is enabled, allowing the mirroring code to almost instantly 'pick up' where it left off by doing a conditional scan of the B-Tree from the root and only following branches with greater serial numbers then its starting serial number, and thus be able to operate in a completely decoupled fashion without having to resort to persistent-stored update queues.
- Hammer includes a utility which through ioctl()s can command the file-system to re-block i-nodes, directory entries, B-Tree nodes, and data. Effectively Hammer is capable of rewriting the entire contents of the file-system while it is live-mounted.
- All Hammer identifiers are 64 bits, including the i-node number. Thus Hammer is able to construct a completely unique i-node number within the context of the file-system which is never reused throughout the life of the file-system. This is extremely important for large scale clustering, mirroring, and remote access. I-nodes are simply B-Tree records like everything else, thus there are no limitations on the number of i-nodes the file-system may contain.
- Media structures are extensible in that just about everything is based on B-Tree records. Adding new record types is trivial and since data is allocated dynamically (even the i-node data), those structures can be enlarged as well.
- Hammer uses 64 bit byte-granular offsets both internally and in the media structures. Hammer has no concept of a sector size but it does use a natural internal buffer size of 16K. A byte-granular offset was chosen not only to make Hammer sector-size agnostic (something that most file-systems are not), but also because it greatly simplified the implementation. Hammer reserves the top 4 bits of its offset identifier as a rough data type (i-node, bulk-data, small-data, etc), and the next 8 bits as a volume identifier.

  Thus Hammer is able to address up to 256 52 bit volumes. That is 256 x

4096 TB or a total of up to 1 Exabyte.

- Hammer CRC checks all major structures and data. The CRC checks are not hierarchical per-say, meaning that updating a low level record does not cause a CRC update to propagate to the root of the file-system tree. All single-structure CRCs are checked on the fly. The block xor integrity check has not been implemented yet as of this writing.
- Clustering and remote-access is a major goal for Hammer, one that is expected to be achieved in a later release. It should be noted that Hammer's transaction identifier and history retention mechanisms are designed with clustering and remote-access in mind, particular with remote cache coherency protocols in mind.

Section 2 – Front-end verses Back-end

Hammer caches all front-end operations, including link count changes, renames, file creation and deletion, and truncation. These are done with small structures in kernel memory. All data modifications are cached via the buffer cache. Hammer includes a bypass mechanism for bulk data, for both reading and writing, allowing it to use cluster_read() for reading and allowing it to commit dirty data directly to the storage media while simultaneously queuing a small structure representing the location of the data on-media to the back-end. The back-end still handles all the B-Tree and other meta-data changes related to the writes. This is actually a fairly complex task as the front-end must not only create these structures, but also access them to handle all nominal file-system activity present in the cache but which has not been flushed by the back-end. Hammer implements an iterator with a full blown memory/media merge to accomplish this.

Link count (and other) changes are cached in the in-memory i-node structure and both new directory entries and deletions from directories are cached as an in-memory record until they can be acted upon by the back-end. Hammer implements a sophisticated algorithm to ensure that meta-data commits made by the back-end are properly synchronized with directory entry commits made in the same cycle, recursively through to the root, so link counts will never be wrong when the file-system is mounted after a crash.

Truncations use a neat algorithm whereby the truncation offset is simply cached by the front-end and copied to the back-end along with the i-node data for action, only when the back-end is ready to sync. A combination of the front-end's understanding of the truncation and the back-end's understanding of the truncation allows the front-end to filter data visibility to make it appear that the truncation was made even though the back-end has not yet actually executed it, and even though while the back-end is executing one truncation the front-end may have done another.

Hammer's front-end can freely manipulate cached i-nodes and records and the back-end is free to flush the actual updates to media at its own pace.

Currently the only major flaw in the Hammer design is that the back-end is so well decoupled from the front-end, it is possible for a large number of records (in the tens of thousands) to build up on the back-end waiting to flush while the front-end hogs disk bandwidth to satisfy user-land requests. To avoid blowing out kernel memory Hammer will intentionally slow down related front-end operations if the queue grows too large, which has the effect of shifting I/O priority to the back-end.

Section 3 – Kernel API

Hammer interfaces with the operating system through two major subsystems. On the front-end Hammer implements the DragonFly VNOPS API. On the back-end (though also the front-end due to the direct-read/direct-write bypass feature) Hammer uses the kernel's buffer cache for both v-node-related operations and for block device operations.

DragonFlyBSD uses a modernized version of VNOPS which greatly simplifies numerous file-system operations. Specifically, DragonFly maintains name-cache coherency itself and handles all name-space locking, so when DragonFly issues a VOP_NRENAME to Hammer, Hammer doesn't have to worry about name-space collisions between the rename operation and other operations such as file creation or deletion. Less importantly DragonFly does the same for read and write operations, so the file-system does not have to worry about providing atomicy guarantees.

Hammer's initial implementation is fully integrated into the DragonFly buffer cache. This means, among other things, that the BUF/BIO API is 64-bits byte-granular rather then block-oriented. It should be noted that Hammer uses dynamically sized blocks, going from 16K records for file offsets less then 1MB and 64K records for offsets >= 1MB. This also means that Hammer does not have to implement a separate VM page-in/page-out API. DragonFly converts all such operations into UIO_NOCOPY VOP_READ/VOP_WRITE ops.

Hammer relies on the kernel to manage the buffer cache, in particular to tell Hammer through the bioops API when it wishes to reuse a buffer or flush a buffer to media. Hammer depends on DragonFly's enhancements to the bioops API in order to give it veto power on buffer reuse and buffer flushing. Hammer uses this feature to passive associate buffer cache buffers with internal structures and to prevent buffers containing modified meta-data from being flushed to disk at the wrong time (and to prevent meta-data from being flushed at all if the kernel panics). This may cause some porting issues and I will be actively looking for a better way to manage the buffers.

Unfortunately there is a big gotcha in the implementation related to the direct-io feature that allows Hammer to use cluster_read(). Because the front-end bypasses Hammer's normal buffer management mechanism, it is possible for vnode-based buffers to conflict with block-device-based buffers. Currently Hammer implements a number of what I can only describe as hacks to remove such conflicts. The re-blocker's ability to move data around on-media also requires an invalidation mechanism and again it is somewhat of a hack.

Finally, Hammer creates a number of supporting kernel threads to handle flushing operations and utilizes tsleep() and wakeup() extensively.

The initial Hammer implementation is not MP-safe, but the design is intended to be and work on that front will continue after the first release. The main reason for not doing this in the first release is simply that building and debugging an implementation was hard enough already (9 months after the design phase was done!). To ensure the highest code quality Hammer has over 300 assertions strewn all over the code base and I decided not to try to make it MP-safe from get-go. Even as work progresses it is likely that only the front-end will be made completely MP-safe. Hammer's front-end is so well decoupled from the back-end that doing this is not expected to be difficult. There is less of a reason to fully thread the back-end, in part because it will already be limited by the media throughput, and also because the front-end actually does the bulk of the media I/O anyway via the bypass mechanism.

Section 4 – The Low-level Storage Layer

As I mentioned in the first section the storage layer has gone through a number of major revisions during the implementation phase. In a nutshell, low level storage is allocated in 8-megabyte chunks via a 2-level free-map. There is no fine-grained bitmap at all. The layer-2 structure (for an 8-megabyte chunk) contains an append offset for new allocations within that chunk, and it keeps track of the total number of free-bytes in that chunk. Various Hammer allocation zones maintain 64 bit allocation indexes into the free-map and effectively allocate storage in a linear fashion. The on-media structures allow for more localization of allocations without becoming incompatible and it is my intention to implement better localization as time goes on.

When storage is freed the free-bytes field is updated but low level storage system has no concept or understanding of exactly which portions of that 8-megabyte chunk are actually free. It is only when the free-bytes field indicates the block is completely empty that Hammer can reuse the storage. This means that physical space on the media will be reused if (A) a lot of information is being deleted, causing big-blocks (8MB chunks) to become complete free or (B) the file-system is re-blocked (an operation which can occur while the file-system is live), which effectively packs it and cleans out all the little holes that may have accumulated.

One huge advantage of not implementing a fine-grained storage layer is that allocating and freeing data is ridiculously cheap. While bitmaps are not necessarily expensive, it is hard to beat the ability to manage 8 megabytes of disk space with a mere 16 bytes of meta-data.

As part of my attempt to look far into the future, the storage layer has one more feature which I intend to take advantage of in future releases. That feature is simply that one can have multiple B-Tree elements pointing to the *same* physical data. In fact one can have multiple B-Tree elements pointing to overlapping areas of the *same* physical data. The free_bytes count goes negative, and so what? That space will not be reused until all the records referencing it go away or get relocated. Theoretically this means that we can implement 'cp a b' without actually copying the physical data. Whole directory hierarchies can be copies with most of their data shared! I intend to implement this feature post-release. The only thing stopping it in the first release is that the re-blocker needs to be made aware of the feature so it can share the data when it re-blocks it.

Remember also that a major feature of this file-system is history retention. When you 'rm' a file you are not actually freeing anything until the pruner comes along and administratively cleans the space out based on how much (and how granular) a history you wish to retain. Hammer is meant to be run on very large storage systems which do not fill up in an hour, or even in a few days. The idea is to slowly re-block the file-system on a continuing basis or via a batch cron-job once a day during a low-use period.

The low level storage layer contains a reservation mechanism which allows the front-end to associate physical disk space with dirty high level data and write those blocks out to disk. This reservation mechanic does not modify any meta-data (hence why the front-end is able to use it). Records associated with the data using this bypass mechanic are queued to the back-end and the space is officially allocated when the records are committed to the B-Tree as part of a flush cycle. The reservation mechanism is also used to prevent freed chunks from being reused until at least two flush cycles have passed, so new data has no chance of overwriting data which might become visible again during crash recovery.

The low level storage layer is designed to support files-system shrinking and expanding, and the addition or deletion of volumes to the file-system. This is a feature that I intend to implement post-release. The two-layer free-map uses the top 12 bits of the 64-bit hammer storage offset as a zone (unused in the free-map) and volume specifier, supporting 256 volumes. This makes the free-map a sparse mapping and adding and removing storage is as easy as adjusting the mapping to only include the appropriate areas. Each volume can be expanded to its maximum size (4096 TB) simply through manipulation of the map. When shrinking or removing areas the free-map can be used to block-off the area and the re-blocker can then be employed to shift data out of the blocked-off areas. As you can see, all the building blocks needed to support this feature pretty much already exist and it is more a matter of utility and ioctl implementation then anything else.

Section 5 – Flush and Synchronization Mechanics

Hammer's back-end is responsible for finalizing modifications made by the front-end and flushing them to the media. Typically this means dealing with all the meta-data updates, since bulk data has likely already been written to disk by the front-end.

The back-end accomplishes this by collecting dependencies together and modifying the meta-data within the buffer cache, then going through a serialized flush sequence. (1) All related DATA and UNDO buffers are flushed in parallel. (2) The volume header is updated to reflect the new UNDO end position. (3) All related META-DATA buffers are flushed in parallel and asynchronously, and the operation is considered complete. The flushing of the META-DATA buffers at the end of the previous flush cycle may occur simultaneously with the flushing of the

DATA and UNDO buffers at the beginning of the next flush cycle.

Thus it takes two flush cycles to fully commit an operation to the media, since a crash which occurs just after the first flush cycle returns cannot guarantee that the META buffers had all gotten to the media, and upon remounting the UNDO buffers will be run to undo those changes.

As Hammer has evolved the move to the use of UNDO records coupled with a back-end which operates nearly independently of the front-end has been done purposefully. The total cost of doing an UNDO sequence is greatly reduced by virtue of not having to generate UNDO records for data, only for meta-data, and by virtue of the complete disconnect between the front-end and the back-end. Efficiency devolves down simply into how many dirty buffers the kernel can manage before they absolutely must be flushed to disk. The use of UNDO records freed the design up to use whatever sort of meta-data structuring worked best.

Section 6 - The B-Tree is a modified B+Tree.

Hammer uses a single, global, modified 63-way B+Tree. Each element takes 64 bytes and the node header is 64 bytes, resulting in a 4K B-Tree node. During the implementation phase I debated whether to move to a 255-way B+Tree so the node would exactly fit into a native 16K buffer, but I decided to stick with a 63-way B+Tree due to the CRC calculation time and the space taken up by UNDO records when modifying B-Tree meta-data.

Hammer uses a somewhat modified B+Tree implementation. One additional element is tagged onto internal nodes to serve as a right-boundary element, giving internal nodes both a left-bound and a right-bound. The addition of this element allows Hammer to cache offsets into the global B-Tree hierarchy on an i-node-by-i-node basis as well as an ability to terminate searches early.

When you issue a lookup in a directory or a read from a file Hammer will start the B-Tree search from these cached positions instead of from the root of the B-Tree. The existence of the right boundary allows the algorithm to seek exactly where it needs to go within the B-Tree without having to retrace its steps and reduces the B-Tree locking conflict space considerably.

For the first release Hammer implements very few B-Tree optimizations and no balancing on deletion (though all the leaves are still guaranteed to be at the same depth). The intention is to perform these optimizations in the re-blocking code and it doesn't hurt to note that Hammer's default history retention mode doesn't actually delete anything from the B-Tree anyway, so re-balancing really is a matter for a background / batch job and not something to do in the critical path.

Section 7 – M-time and A-time updates

After trying several different mechanics for mtime and atime updates I finally decided to update atime completely asynchronously and to update mtime as part of a formal back-end flush, but in-place instead of generating a new inode record and also without generating any related UNDO records. In addition, the data CRC does not cover either field so the B-Tree element's CRC field does not have to be updated. This means that upon crash recovery neither field will be subject to an UNDO. The absolute worst case is that the mtime field will have a later time then the actual last modification to the file object, which I consider acceptable.

Simply put, trying to include atime and mtime (most especially atime) in the formal flush mechanic led to unacceptable performance when doing bulk media operations such as a tar.

The mtime and atime fields are not historically retained and will be locked to the ctime when accessed via a snapshot. This has the added advantage of producing consistent md5 checks from tar and other archiving utilities.

Section 8 – Pseudo-file-systems, Clustering, Mirroring

Hammer's mirroring implementation is based on a low level B-Tree element scan rather then a high-level topological scan. Nearly all aspects of the file-system are duplicated, including inode numbers. Only mtime and atime are not duplicated. Hammer uses a queue-less, incremental mirroring algorithm.

To facilitate mirroring operations Hammer takes the highest transaction id found at the leafs of the B-Tree and feeds it back up the tree to the root, storing the id at each internal element and in the node header. The mirroring facility keeps track of the last synchronized transaction id on the mirroring target and does a conditional scan of the master's B-Tree for any transaction id greater or equal to that value. The feedback operation makes this scan extremely efficient regardless of the size of the tree; the scan need only traverse down those elements with a larger or equal valued transaction id.

To allow inode numbers to be duplicated on the mirroring targets Hammer implements pseudo-file-system (PFS) support. Creating a PFS is almost as easy as creating a directory. A Hammer file-system can support up to 65536 PFSs.

Thus Hammer can do a full scan or an incremental scan, without the need for any queuing, and with no limitations on the number of mirroring targets. Since no queuing is needed all mirroring operations can run asynchronously and are unaffected by link bandwidth, connection failures, or other issues. Mirroring operations also have no detrimental effect on the running system, other then the disk and network bandwidth consumed. Data transfer is maximally efficient.

Mirroring operations are not dependent on Hammer's history retention feature but use of history retention does make it possible to guarantee more consistent snapshots on the mirroring targets. If no history retention occurs on the master or if the mirroring target has gotten so far behind that the master has already started to prune information yet to be copied, the solution is for the mirroring code on the target to simply maintain a parallel scan of the target's B-Tree given key ranges supplied by the source. These keys ranges are constructed from the transaction ids propagated up the tree so even in no-history mode both the master and the target can short-cut their respective B-Tree scans.

Because all mirroring operations are transaction-id based Hammer maintains a PFS structure controlled by the mirroring code which allows a snapshot transaction id to be associated with each mirroring target. This snapshot id is automatically used when a PFS is accessed so a user will always see a consistent snapshot on the mirroring target even though ongoing mirroring operations are underway. Similarly, when mirroring from the master, the last committed transaction id is used to limit the operation so the source data being mirrored is also a consistent snapshot of the master.

Hammer is part of a larger DragonFly clustering project and the intention is to evolve the file-system to a point where multi-master replication can be done. This feature clearly will not be available in the first release and also requires a great deal of OS-level support for cache coherency and other issues. Hammer's currently mirroring code only supports a single master, but can have any number of slaves (mirroring targets). Ultimate up to 16 masters will be supported.

Section 9 – The File-system is Full! (and other quirks)

What do you do when the default is to retain a full history? A 'rm' won't actually remove anything! Should we start deleting historical data?

Well, no. Using Hammer requires a somewhat evolved mind-set. First of all, you can always mount the file-system 'nohistory', and if you do that 'rm' will in fact free up space. But because space is only physically reusable when an eight-megabyte chunks becomes fully free you might have to remove more then usual, and frankly you still need to run the hammer utility's re-blocker to pack the space.

If you are running in the default full-history retention mode then you need to set up a pruning regimen to retain only the history you want. The 'hammer softprune' command does the trick neatly, allowing you to control what snapshots you wish to retain simply by managing a set of softlinks in a directory.

Hammer is not designed to run on tiny amounts of disk space, it is designed to run on 500GB+ disks and while you can run it spaces as small as a gigabyte or two, you may run into the file-system-full issue more then you would like if you do. But with a large disk space management takes on a new meaning. You don't fill up large disks. Instead you actively manage the space and in the case of Hammer part of that active management is deciding how much history (and at what granularity) you want to retain for backup, security, and other purposes.

Active management means running the hammer softprune and hammer reblock family of commands in a daily cron job to keep things clean. The hammer utility allows you to run these commands in a time-limited fashion, incrementally cleaning the file-system up over a period of days and weeks. I would like to implement some sort of default support thread to do this automatically when the file-system is idle, but it will not be in by the first release. The hammer utility must be run manually.

If there are particular files or directory hierarchies that you do not wish to have any history retention on Hammer implements a chflags flag called 'nohistory' which will disable history retention on a file or directory. The flag is inherited from the parent directory on creation. Files with this flag set will not show up properly in snapshots, for obvious reasons.

Hammer directories use a hash key + iterator algorithm and creates one B-Tree element per directory entry. If you are creating and deleting many versions of the same file name the directory can build up a large number of entries which effectively have the same hash key and all have to be scanned to locate the correct one. The first release of Hammer does not address this potential performance issue other then to note that you might want to run the hammer softprune utility more often rather then less in such situations.

Section 10 – History Retention and Backups

In Hammer accessing a snapshot or prior version of a file or directory is as simple as adding a "@@0x<64_bit_transaction_id>" extension to the file or directory name. You can, in fact, CD into a snapshot that way. Most kernels do a file-system sync every 30-60 seconds which means, by default, Hammer has an approximately one-minute resolution on its snapshots without you having to lift a finger. Hammer's history retention is based on data committed to the media so cached data, such as found in the buffer cache, creations, deletions, renames, appends, truncations, etc... all of that is *not* recorded on a system-call by system-call basis. It is recorded when it is synced to the media.

Even if you do not wish to retain much history on your production systems you still want to retain enough so you can create a softlink to snapshot the production system, and use that as your copy source for your backup. Hammer's history retention mechanism is straight out the single most powerful feature of the file-system.

Because Hammer retains a full history of changes made to the file-system backing it up on your LAN or off-site is more a matter of replication then anything else. If your backup box is running Hammer all you have to do is replicate the masters on the backup daily, *to the same directory structure*. You do not need to use the hard-link trick, or make additional copies, just overwrite and create a soft-link to each day's snapshot. The rdist or cpdup programs are perfect for this sort of thing, just as long as the setup leaves unchanged files alone. The backup box's drive space can be managed by selectively deleting softlinks you do not wish to retain (making your backups more granular) and then running the 'hammer prune' command to physically recover the same.

Backups can also be maintained via the mirroring mechanism. You can create a softlink for each incremental mirroring operation and manage the history retention on the mirroring targets independent of the history retention on the master.

Section 11 – Data Integrity and Snapshots

Hammer's CRCs do a good job detecting problems. There is a field reserved for a whole-file data CRC to validate that all expected B-Tree elements are present, but it has not been implemented for the first release.

Hammer's snapshots implement a few features to make data integrity checking easier. The atime and mtime fields are locked to the ctime when accessed via a snapshot. The st_dev field is based on the PFS shared_uuid setting and not on any real device. This means that archiving the contents of a snapshot with e.g. 'tar' and piping it through something like md5 will yield a consistent result that can be stored and checked later on. The consistency is also retained on mirroring targets. If you make archival backups which retain ctime then snapshot access will also produce a consistent result, though the st_dev field will be different unless you also make the archival backup in a PFS with the same shared-uuid setting. Strict mirroring is thus useful, but not required to create consistent snapshots of important data.

Hammer does not implement single-image replication or redundancy. The intention is to implement redundancy via multi-image replication, what is currently single-master/multi-slave mirroring and will eventually become multi-master/multi-slave mirroring. Now, of course in any large scale system even the single images would be backed by a RAID array of some sort, but Hammer does not try to implement block-level redundancy on its own. Frankly it is a bit dangerous to rely on block level redundancy when a large chunk of file-system corruption issues are due to software bugs that block level redundancy cannot actually detect or fix. A replication architecture based on a logical layer (B-Tree elements or topological replication), backed by redundant storage, and a backup system with live access to incremental backups is an absolute requirement if you care about your data at all.

One can only trust file-system data integrity mechanics to a point. The ultimate integrity check will always be to serialize an archive using a topological scan (like 'tar') and pipe it to something like md5, or sha1, or things of that ilk.

Section 12 – Performance Notes (July 2008 release)

In its first release Hammer's performance is limited by two factors. First, for large files each 64K data block requires one 64 byte B-Tree element to manage. While this comes to a tiny percentage space-wise, operations on the B-Tree are still considerably more expensive then equivalent operations would be using a traditional block-map. Secondly, the current dead-lock handling code works but is fairly inefficient and the flusher threads often hit dead-locks when updating large numbers of i-nodes at once. A deadlock is not fatal, it simply causes the operation to retry, but it does have an impact on performance.

Hammer does have a larger buffer cache footprint verses a file-system like UFS for the same data set. This tends to impact reads in random access tests only if the active data-set exceeds available cache memory. One would expect it to impact writes as well but modifications are detached to such a high degree that Hammer's random write performance both in a cache-blown-out and cache-not-blown-out scenario tends to be significantly better. Most of the overhead is due to Hammer's use of larger buffer cache buffers then UFS.

Hammer's cpu overhead is roughly equivalent to UFS, but it took a few tricks to get there and those tricks do break down a little when it comes to updating large numbers of i-nodes at once (e.g. atime updates). In the first release users with bursty loads might notice bursty cpu usage by Hammer's support threads.

As of the first release Hammer does have a performance issue when partially rewriting the contents of a directory. For example, when creating and deleting random files in a directory. If the data-set gets large enough a random access load will both blow out the caches and prevent Hammer from doing a good job caching meta-data, leading to poor performance. The performance issues for the most part go away for any data stored in the long-term as the re-blocker does a good job de-fragmenting it all. Short-lived files are also unaffected since they typically never even reach the media. Directories which operate like queues, such as a sendmail queue, should also be unaffected.

Section 13 - Porting

If you've slogged through this whole document then you know that while the first release of Hammer is very complete, there are also a ton of very cool features still in the pipeline. I intend to support porting efforts and I'm gonna say that doing so will probably require a lot of work on the source code to separate out OS-specific elements. The only way to really make Hammer a multi-port file-system is to focus on splitting just four files into OS-specific versions. The whole of “hammer_vnops.c”, the whole of “hammer_io.c”, a port-specific header file, and an OS-specific source file supporting a Hammer-API for mount, unmount, and management of the support threads (things like tsleep, wakeup, and timestamps).

If we do not do this then porters are likely going to be left behind as work continues to progress on Hammer. I am looking into placing the code in its own repository (something other then CVS) so it can be worked on by many people.

Here the big porting gotchas people need to be on the lookout for:

- Inode numbers are 64 bits. I'm sorry, you can't cut it down to 32. Hammer needs 64.
- Hammer uses 64 bit byte offsets for its buffer-cache interactions, but accesses will always be in multiples of 16K (and aligned as such).
- Hammer uses 16K buffers and 64K buffers within the same inode, and will mix 16K and 64K buffers on the same block device.
- Hammer's direct-io bypass creates a lot of potential conflicts between buffer cache buffers of differing sizes and potentially overlapping the same physical storage (between a buffer cache buffer associated with a front-end vnode and one associated with a back-end block device). Lots of hacks exist to deal with these.
- Hammer's DragonFly implementation uses advanced bioops which allows Hammer to passively return the buffer to the kernel, but still have veto power if the kernel wants to reuse or flush the buffer.
- Hammer uses DragonFly's advanced VNOPS interface which uses a simplified name-cache-oriented API for name-space operations.

- Hammer expects the kernel to prevent name-space and data-space collisions and to handle atomicy guarantees. For example if you try to create a file called “B” and also rename “A” to “B” at the same time, the kernel is expected to deal with the collision and not dispatch both operations to the Hammer file-system at the same time.

  Kernels which do not deal with atomicy and name-space guarantees will need to implement a layer TO guarantee them before dropping into Hammer's VNOPS code.