Dynamic Vnodes - Design and Implementation

Aju John
Digital Equipment Corporation

Abstract

Dynamic vnodes make the UNIX kernel responsive to a varying demand for vnodes, without a need to rebuild the kernel. It also optimizes the usage of memory by deallocating excess vnodes. This paper describes the design and implementation of dynamic vnodes in DEC OSF/1 V3.0. The focus is on the vnode deallocation logic in a Symmetric Multi-Processing environment.

Deallocation of vnodes differs from the familiar concept of dynamically allocated data structures in the following ways: the legacy name-cache design implicitly assumes that vnodes are never deallocated, and the vnode free-list needs to cache unused vnodes effectively.

1. Introduction

Digital's DEC OSF/1 uses the Virtual File System *(VFS)* [OSF93, Langerman90] to support multiple file-system types. VFS is a subsystem that provides a uniform means of accessing the files in the system. *Vnodes* [Kleinman86] are an integral part of the VFS layer and are used to store the file index information structure of the underlying file-system, such as a UFS inode [Bach86, LoVerso91]. A *vnode table* is a static array of vnode structures that is generally pre-allocated at boot time. The dynamic vnodes implementation was part of a larger scalability project whose objective was to eliminate the static file-system tables in DEC OSF/1.

Dynamic vnodes free the system from the limits imposed by the static vnode table and prevent memory from being taken up by excess vnodes. Making a table dynamic is a familiar process. It involves allocating each element when there is a demand and, when it is no longer in use, deallocating it to reclaim resources. However, in the case of vnodes, the task is more complex because it is necessary to:

- ensure that a vnode will not be referenced by the name-cache code after it has been deallocated
- strike a balance between caching vnodes on the free-list and deallocating excess vnodes, depending on the system's current demand for vnodes

The discussion in this paper covers the design and implementation of dynamic vnodes. The allocation policy is a basic enhancement over the design used in the Open Software Foundation's reference version 1.1.3 of OSF/1 source code. However the reference version did not support vnode deallocation, so it was not possible to reclaim the memory used by vnodes. The implementation of dynamic vnodes resolves all the challenges associated with deallocating vnodes, making it possible to grow and shrink the number of vnodes based on demand.

In addition, the implementation ensures that the deallocation of vnodes does not impact the overall performance of the system. This is done by making the implementation sensitive to demand for vnodes. The implementation also improves the chances of reusing useful file system data that are cached in the free vnodes on the free-list.

This paper will describe the design and show its integration into the existing vnode management structure. The design is easy to integrate into virtually any kernel that uses 4.4BSD-style or OSF/1-style VFS, that is, a kernel that allows vnodes from one file-system type to be re-allocated [OSF93] to another type. The design uses a timestamp based technique to deal with the legacy name-cache design. This technique can be applied in general to deal with legacy data structures that may have references to deallocated memory.

2. Motivation for dynamic vnodes

The need for a kernel that can scale its responses to the demand for vnodes was the motivating factor for this design and implementation. In short, the kernel should be able to dynamically respond to varying demand for vnodes, without a need to be rebuilt when it runs out of vnodes. At the same time, it should release memory taken up by excess free vnodes. From a performance perspective, the deallocation design should be able to deal with references to deallocated vnodes in the name-cache table, without having to search all name-cache entries. In addition, the deallocation policy has to be sensitive to the system's demand for vnodes. It should retain an optimal number of vnodes on the free-list for effective caching and balance it with the amount of memory consumed by vnodes.

2.1 Limitations of a static vnode table

A kernel based on a static vnode table can handle demand for vnodes up to a maximum determined by the size of the table. Such a kernel would have to be re-configured and re-built whenever the working set of vnodes exceeds the table size. Several versions of UNIX in the market, including DEC OSF/1 Version 2.0 and earlier, use a static vnode table. Its size was determined at startup, usually based on the value of maximum number of users that is set for the system. Vnodes were assigned from this table when needed and returned to be re-used or reclaimed when they were no longer needed.

When processes on such systems request an additional vnode over the table limit, the processes are suspended indefinitely or exit with an error message such as "`Vnode table full`". This problem became increasingly common in environments that had a large number of users.

In such a situation, the solution was to rebuild the kernel with a larger value for the maximum number of users, which in turn would create a larger vnode table, and then reboot with the new kernel. On the other hand, such a demand for vnodes could only be for a short duration. So, a kernel with a huge static vnode table might be able to handle most types of vnode demands when they happen, but this would waste memory resources during periods of low vnode demand. Consider this example, where the current size of a vnode in DEC OSF/1 V2.0 is 488 bytes: A kernel that has a vnode table that holds 20,000 vnodes will take up about 10MB of memory, regardless of the number of vnodes used.

The Open Software Foundation partially addressed this problem in the OSF/1 reference code by replacing the static vnode table and allocating vnodes when new ones are needed. This eliminated the problem of the system running out of vnodes. However, in the OSF/1 reference code, once a vnode is allocated, it is not deallocated. The Open Software Foundation probably decided to defer implementing the vnode deallocation in OSF/1 reference code due to the inherent design of the vnode name-cache, which implicitly assumes that vnodes are never deallocated.

2.2 Problems with allocation-only policy

Earlier internal releases of DEC OSF/1 had the vnode allocation-only scheme, with the added functionality to cache vnodes for a certain duration on the free-list. If the vnode at the head of the free-list is fairly new, a new vnode is allocated when needed, instead of recycling the one at the head of the free-list. Soon, it was noticed that a significant amount of memory was being dedicated to the ever increasing number of vnodes and the buffers associated with the vnodes.

DEC OSF/1 tries to cache the vnodes on the free-list for a while, so that any of them can be quickly re-activated if a vnode lookup operation succeeds. The advantage of caching vnodes on the free-list for at least a fixed duration is that it prevents the cached vnodes from being flushed by simple operation such as *ls -lR(1)* [DEC-RP]. The LRU (Least Recently Used) nature of the free-list combined with the caching of free vnodes for a fixed duration causes the free-list size to grow during such operations. This allows the frequently used vnodes to get re-activated instead of re-used, while the LRU nature of the free-list pushes the other vnodes to the head of the free-list for re-use. Re-activating more vnodes from the free-list is a definite performance win.

However, if vnodes cannot be deallocated, operations such as *ls -lR* can use up a significant amount of memory in the form of newly allocated vnodes. A sudden, but not too uncommon spike of vnode requests caused by programs such as a *find(1)* [DEC-RP] operation can also take up a significant amount of memory for vnodes.

Consider a mid-sized machine such as the *Digital AlphaServer 2100 4/200.* Under low vnode demands, the system typically uses about a 1000 vnodes. However, it can allocate up to a maximum of 41,000 vnodes (Section 4.1) that takes up 5% of the 384MB of memory in a typical configuration. Over 19MB of memory will be taken up by vnodes if the system is unable to deallocate excess vnodes.

As the preceding example illustrates, there was a pressing requirement to design and implement truly dynamic vnodes that could be allocated and deallocated based on demand. The deallocation policy had to be robust enough to deallocate excess free vnodes, while maintaining an effective cache of frequently used vnodes on the free-list.

2.3 The name-cache design issue

VFS uses the name-cache or the Directory Name Lookup Cache *(DNLC)* [Langerman90] to improve the pathname translation performance by providing fast lookups of frequently used file names. The name-cache poses the biggest challenge to vnode deallocation. The name-cache is indexed by a hash value on the pair *vp, name*, where *vp* is the vnode that refers to the directory that contains *name.* Once a reference to a vnode gets cached in the name-cache, a lookup operation on that vnode can happen any time in the future. If the lookup operation picks up the reference to the vnode after it has been deallocated, the kernel will crash trying to access invalid memory. When memory allocated to a vnode is deallocated, all cache references to that vnode must be removed from the name-cache.

The process of ensuring that all references to a vnode are removed is a challenging one for many reasons. Several different entries in the name-cache may refer to the same vnode. The name-cache is not indexed just by the vnode alone and the mapping between names and vnodes is a many-to-one type. It is prohibitively expensive to walk the entire name-cache, looking for every vnode that is targeted for deallocation, to ensure that all cache references to that vnode have been removed. Re-designing the name-cache to maintain a linked list of all the entries that correspond to a vnode reduces performance in a multiprocessing environment (Section 4.3.2) and introduces race conditions.

Therefore, a low-overhead time-stamp based technique was designed to deal with references to deallocated vnodes in the name-cache table.

3. Dynamic vnodes

Vnodes that can be allocated on demand and deallocated to return the resources used by them are termed *dynamic vnodes*.

Dynamic vnode design discussed here can be used to modify any kernel that has the 4.4BSD-style or the OSF/1-style VFS layer. The chief characteristic of this type of VFS layer is that vnodes are not bound to any one file-system type. In such an implementation, a vnode can get recycled from one type to another after it has been cleaned. Before cleaning it, the *vgone()* operation eliminates all activity associated with the vnode. The *vclean()* routine cleans the vnode by doing the file-system specific reclaim operation on it. This operation disassociates the file-system dependent data from the vnode before re-use. Other implementations of VFS, such as the one in SunOS, are different from the type described here.

3.1 Design Principles

Dynamic vnodes were designed so that the following conditions were always adhered to:

- dynamically configurable limits
- low overhead
- re-use all of the existing VFS logic and code
- multi-processor safe

Dynamically configurable limits allow any limits in the design to be changed by the administrator on a running kernel, totally transparent to all other users in the multi-user mode.

Low overhead deallocation was a necessity to ensure that the overall performance of a kernel with dynamic vnodes would be as close as possible to the kernel with static vnodes or one with dynamic allocation alone. It was also necessary to handle name-cache entries of vnodes that were deallocated, without incurring the high cost of search operations.

The emphasis was to adhere to the original VFS code and to integrate the deallocation code into the existing code base. This has two advantages. It will still make use of all the time-tested VFS code that was already in place and, if needed, the new code can be easily turned off or excluded, making the system just as it was before the change. In fact, one of the design criteria was to make it possible to switch the kernel from a deallocation mode to a no-deallocation mode at run time. This was essential to understand and improve the performance of the deallocation code.

The new design had to be safe for use in a symmetric multiprocessing environment, without any race conditions between the deallocation and the lookup code paths. At the same time, the design had to avoid being riddled with locks, which would reduce performance.

4. Design and Implementation

Dynamic vnode design is discussed in two parts. Allocation of vnodes is the first part, which is an enhancement to the design from the Open Software Foundation's reference version 1.1.3 of OSF/1 source code. This will be briefly discussed with emphasis on the enhancements. The second part is the vnode deallocation design, which will be discussed in detail.

4.1 Allocation of vnodes

The allocation scheme uses the *power-of-two* [McKusick88] allocator to allocate vnodes when needed. At system start-up, there are no vnodes in the system. The first vnode is dynamically allocated in the VFS initialization routine. A new function called *initnewvnode()* allocates a new vnode and returns its address after initializing all of the fields. The *total_vnodes* count is incremented with each vnode that gets allocated.

To make optimum use of memory while using the power of two allocator, the vnode structure was re-sized to be under 512 bytes. This was done by rearranging the fields so that the structure was packed tight and also by eliminating flag fields and lock fields that could be merged with other existing fields. In order to keep the code intact, new macros were written to make the vnode structure changes transparent to the rest of the VFS code that uses it.

The following terms are important to this discussion:

- **max-vnodes:** the maximum number of vnodes that can be present in the system.
- **minimum-free-vnodes:** the minimum number of free vnodes that should be present in the system for vnode caching.
- **vnode-age:** a value in seconds that is used to determine if a vnode at the head of the free-list can be recycled rather than allocating a new vnode.

Although in theory vnodes can be allocated without an upper bound, a soft limit called *max-vnodes* was introduced to prevent runaway allocations. The value of *max-vnodes* is set when the system boots, depending on the amount of memory in the system. By default, the *max-vnodes* value is set to the number of vnodes that can fit in five percent of the system memory.

The VFS *getnewvnode()* function allocates a new vnode by calling *initnewvnode()* if the number of free vnodes is less than *minimum-free-vnodes* or if the vnode at the head of the free-list is not older than *vnode-age*. This gives the vnode at the head of the free-list a chance to be reclaimed and makes vnode caching more effective. Thus, on a busy system, the vnode free-list tends to be longer. A new vnode is always allocated if the number of free vnodes is less than *minimum-free-vnodes*.

The values *of max-vnodes, minimum-free-vnodes,* and *vnode-age* can be changed dynamically on a running system using a kernel debugger or by using the *sysconfig(8)* [DEC-RP, DEC-ST] utility.

4.2 Deallocation of vnodes

When the number of free vnodes in a system is fairly high and the rate of usage of vnodes in the system is fairly low, vnode deallocation routines release the memory held by the excess vnodes back to the system.

4.2.1 Name-cache complications

With most system data structures, deallocation would have been as simple as freeing the memory assigned to the vnodes in an LRU fashion. Because the vnode free-list is maintained in an LRU order, vnode deallocation would appear as a simple task of freeing vnodes closer to the head of the list.

However, there are references to vnodes in the name-cache that are used for quick pathname translation [OSF93, Langerman90]. The design of the name-cache assumed that references to vnodes are always valid and did not take into account the fact that memory allocated to a vnode can get freed. It depended on the vnode identifier field in the vnode called *v_id* to determine whether the reference to the vnode is valid. When a vnode is deallocated, it is not economical to search the name-cache for obsolete entries to get rid of them. This is because several different name-cache entries may refer to a single vnode and the number total number of entries in the name-cache can be very large. On the other hand, when a lookup operation returns a reference to a vnode, that vnode should not be a deallocated vnode. A simple cache-purge operation alone would not suffice because the method it uses to invalidate a reference to a vnode is to increment the vnode's generation number. If the vnode is going to be deallocated, the contents of that vnode will be lost and incrementing the generation number will not work.

The problem is compounded by the fact that the logic had to be safe for a multiprocessing environment. When deallocating, races had to be avoided without getting into a deadlock [Bach84]. The following races are possible if vnode deallocation is introduced:

- a vnode can get deallocated between a vnode-lookup operation and a vnode-get *vget()* operation.
- a vnode-lookup operation can race with the deallocation operation on a particular vnode.
- name-cache purge operations for vnodes due to umount operations can race with the deallocation operations for the vnodes.
- vital counters can get corrupted if allocation races with deallocation, resulting in system errors or memory leaks due to lost vnodes.

Our solution implements vnode deallocation by using time-stamps based on the value of *lbolt*.

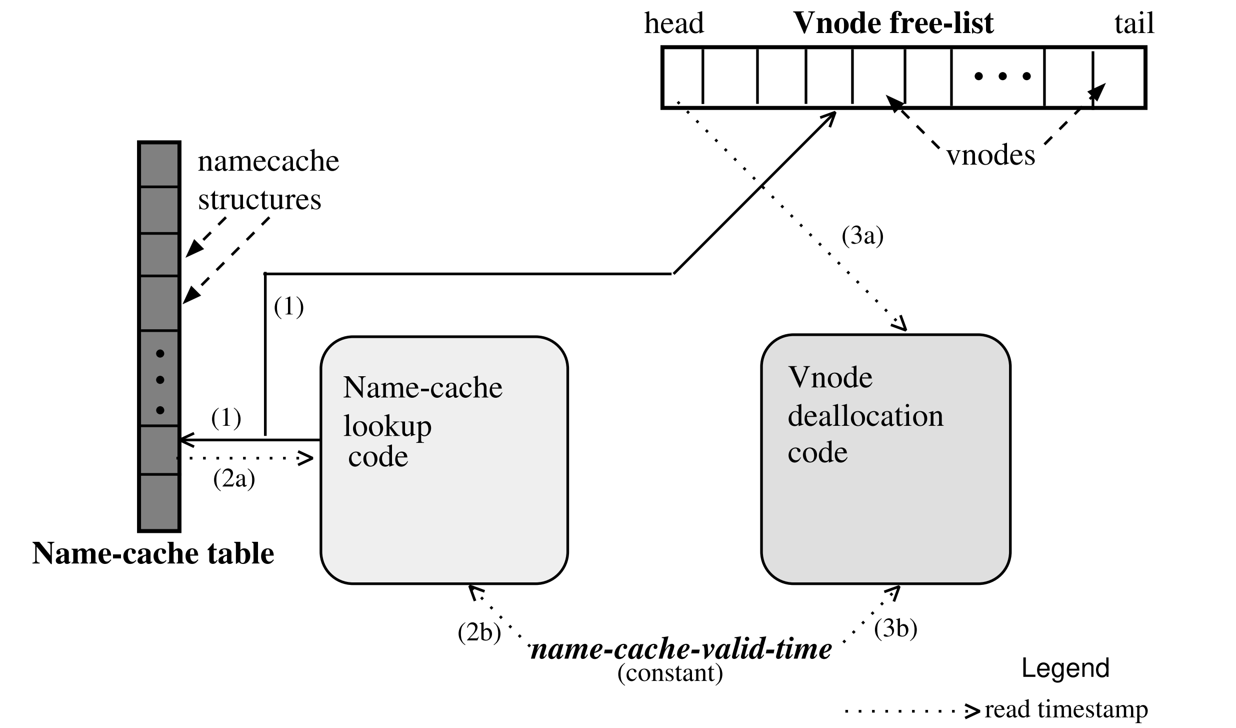

Figure 1: Vnode deallocation: ***The essence of the design***

4.2.2 Timestamps using lbolt

The kernel variable called *lbolt* stores the number of hard-clock interrupts since boot. Using a very reliable timestamp mechanism was critical to the design. The system-time was not used for timestamps because it can be changed on a running system. The main reason for using *lbolt* is that its value is guaranteed to increment, and never to decrement. It is also easy to convert the lbolt value into seconds by a low-overhead shift operation based on the system's frequency.

4.2.3 Design of the deallocation logic

The deallocation code introduces the following kernel variable, which plays a key role in the design:

- **name-cache-valid-time:** name-cache entries with timestamps older than this value are considered stale and are discarded. Vnodes that have timestamps older than this value are chosen for deallocation.

Vnodes from the head of the free-list that have not had any name-cache activity in the last *name-cache-valid-time* seconds are considered for deallocation. During a lookup operation, if the name-cache entry is older than *name-cache-valid-time*, the associated vnode may have been deallocated, so its address is not returned. The value of this variable can be set at boot-time, but cannot be tuned on a running kernel.

Figure 1 illustrates the working of the logic. Timestamp fields are present in the vnode and the name-cache structures. In the figure, the timestamp update process is represented by solid lines, while timestamp read operations are represented by dotted lines. The name-cache lookup code is invoked whenever there is a need to add or to look up an entry in the *name-cache table*. When an entry gets added, that entry and the corresponding vnode are time-stamped (1). When a sought entry is found in the table during a lookup operation, its time-stamp is checked (2a) to see if the entry is older than *name-cache-valid-tim*e (2b). The lookup operation returns the entry if it is not older than *name-cache-valid-time* and the timestamps of the entry and the corresponding vnode are updated (1) atomically. If the name-cache entry is older than *name-cache-valid-time,* it is purged and the lookup operation returns a failure because the corresponding vnode could have been deallocated.

Figure 1 also illustrates the vnode deallocation code that inspects the timestamp on the vnode (3a) at the head of the free-list that is being considered for deallocation. It is compared (3b) with the value of *name-cache-valid-time*. One of the conditions necessary to deallocate a vnode is to have its timestamp older than *name-cache-valid-time*.

Clearly, the vnode deallocation code depends on timestamps as a part of its design. New fields were added to the vnode and the name-cache structures for this purpose.

```
struct vnode {
    /* new fields only */
    u_int  v_ncache_time;
    u_int  v_free_time;
    u_int  v_cache_lookup_refs;
 }
```

Every vnode has two new timestamp fields, one to store the time it was put on the free-list *(v_free_time)* and the other to store the time of the last name-cache activity for that vnode *(v_ncache_time)*. The *v_cache_lookup_refs* is a counter that keeps track of pending *vget* operations on vnodes that have been successfully looked up in the name-cache.

```
struct namecache {
    /* new field only */
    u_int nc_time;
}
```

Every name-cache entry has a new timestamp field *(nc_time)* to store the last name-cache activity for that entry.

4.2.3.1 Resolving the name-cache issue

When name-cache activity such as a cache-lookup or cache-enter operation happens, the name-cache-activity field in the vnode *(v_ncache_time)* and the *nc_time* field in the name-cache entry are updated simultaneously with the current time index based on the value of *lbolt*. Whenever a vnode is put on the free-list, the current lbolt-timestamp value is stored in the vnode's *v_free_time* field.

The free-list time field *(v_free_time)* is used by *getnewvnode()* to determine if a vnode at the head of the free-list needs to be cached for a longer time or if it can be recycled. The last name-cache activity timestamp field on a vnode *(v_ncache_time)* is used by the deallocation code to determine if that vnode can be deallocated. If its value is older than *name-cache-valid-time*, that vnode is a candidate for deallocation.

Thus, the deallocation code guarantees only to deallocate a vnode if there was no name-cache activity for that vnode for a period of time determined by the value of *name-cache-valid-time*. The name-cache code depends on this guarantee made by the deallocation code to decide whether to send back references to vnodes for successful cache-lookup operations. If the name-cache activity timestamp on a cache entry that was successfully looked up is older than the predetermined *name-cache-valid-time,* the vnode reference is not returned. Instead, that name-cache entry is deleted and a counter for such hits is incremented. In this way, references to deallocated vnodes can be avoided without having to examine every entry in the name-cache for possible deletion. By counting the number of bad cache hits due to an old timestamp, the loss in performance of the name-cache can be determined. This helped in tuning the value of *name-cache-valid-time* to make the cache-lookup performance loss under 0.1% for different types of load conditions, as shown in Section 5.

4.2.4 When to deallocate a vnode:

A good time to check for excess free vnodes on the free-list is when vnodes are being added to the tail of the free-list. If the number of free vnodes in the system is greater than the value of *minimum-free-vnodes*, the vnode deallocation function *vdealloc()* gets called whenever a vnode is released onto the free-list by *vrele()*.

The *vdealloc()* function attempts to deallocate vnodes from the head of the free-list, which is protected by a free-list lock. For every vnode selected for deallocation, its *v_cache_lookup_ref*s field is examined to see if there are any *vget()* operations pending after a successful cache-lookup operations for that vnode. If so, this vnode can get referenced any time and is not a good choice for deallocation. Next, the timestamp field *v_ncache_time* for the last name-cache activity is examined. If any cache-lookup activity has happened in the last *name-cache-valid-time* seconds, the deallocation code must honor the guarantee and return the vnode to the head of the free-list.

To reduce the number of times the free-list-lock gets called, it tries to deallocate vnodes in groups. To defray the cost of deallocation across several *vrele()* operations, only a fraction of the excess vnodes are deallocated by each *vdealloc()* call.

Every *getnewvnode()* operation also inspects the vnode it takes off the free-list and deallocates it if it meets the deallocation criteria. To prevent *getnewvnode()* from getting too busy deallocating old vnodes, a configurable number sets a limit on the number of vnodes that it can deallocate during each call.

4.2.5 Tuning deallocation to system load.

A special vnode called *marker-vnode* is injected into the free-list chain. The special vnode is never removed, but it cycles constantly through the vnode free list from the tail to the head. When it reaches the head, the *getnewvnode()* function detects it and puts it at the tail of the free-list, where it will continue traveling toward the head of the free-list. Each time it is put at the end of the free-list, it is timestamped. By examining the marker vnode, the deallocation code can determine the cycle-rate of free vnodes. If the vnodes in the free-list are cycling at a rapid rate, the deallocation code will back off until the vnode activity on the free-list is slow. When the cycle-rate is not too rapid, the deallocation code can consider deallocating vnodes from the head of the free-list.

Vnode deallocation is temporarily turned off for a tunable period of time if any of the following conditions are met:

- deallocation attempts fail more than a pre-determined number of times per batch attempt. Deallocation will not happen unless either a pre-determined number of vnodes have been removed from the head of the free-list for re-use or a pre-determined amount of time has elapsed.
- the number of free-vnodes falls below the value of *minimum-free-vnodes.*
- the marker-vnode is cycling through the free-list at a rate faster than once per *vnode-age* seconds.

4.2.6 Handling SMP Races

This section describes how the races encountered in a Symmetric Multi-Processing (SMP) environment were handled by the design.

4.2.6.1 Handling the *vget()* race

The *v_cache_lookup_refs* field in the vnode is used to keep track of any successful cache-lookup references for that vnode. Normally, a successful cache-lookup operation is followed by a *vget()* operation to remove the vnode from the free-list. In between the cache-lookup and the *vget()* operations, the vnode should not get deallocated. So, if there has been a successful cache-lookup operation, the *v_cache_lookup_refs* field value is incremented, which serves as a hint to the deallocation code not to deallocate that vnode. This prevents the race between vnode deallocation and *vget()*. Later, when the vget() operation happens on the vnode, the counter is decremented. In the kernel, the *vget()* function calls after cache-lookup operations are replaced by a macro called *vget_cache()*, which causes the lookup reference count to be decremented. The *v_cache_lookup_refs* field in the vnode is also protected by the same global simple-lock used to protect the timestamp fields. For some reason, if any code in the kernel decides not to do a *vget()* operation on a vnode after a successful cache-lookup operation, it should call a function called *abort-vget()*, which just decrements the cache-lookup-reference count for that vnode. If this is not done, the vnode will not be deallocated. The *v_cache_lookup_refs* field is also incremented by *iget()* when it finds a vnode it needs on the free-list using *vget().* This prevents the vnode from being deallocated while *iget()* is holding a reference to it.

4.2.6.2 Handling the cache-lookup race

A possible race condition exists where a vnode can get deallocated while it is being looked up in the name-cache. To prevent this from happening, the simple-lock that is used to protect the *v_cache_lookup_refs* field in the vnode has to be acquired by the deallocation operation and the cache-lookup operation. This prevents the cache-lookup operation and the vnode deallocation operation from racing with each other. If the cache-lookup operation succeeds in acquiring the lock first, it increments *v_cache_lookup_refs* field in the vnode as discussed earlier (Section 4.2.6.1). If deallocation code succeeds in acquiring the simple-lock, the cache-lookup operation will have to wait until the lock is released by the deallocation code. The subsequent timestamp check in the cache-lookup operation will ensure that the vnode will not be referenced if it has been deallocated.

4.3 Other Avenues explored

During the process of design and implementation, several variations of the design were considered and a few of them were prototyped. This section discusses some of them and reasons why they were not used.

4.3.1 Name-cache references in the vnode.

An earlier implementation of the design did not use the *name-cache-valid-time* method. Instead, every time a name-cache entry was made for a vnode, the address of the name-cache entry was stored inside the vnode structure. Vnodes were deallocated based on the age of the vnode on the free-list. When the vnode was being deallocated, it had a reference to the name-cache entry corresponding to it, which could be purged at the same time. This design had a major drawback because a vnode could have multiple name-cache entries, while the vnode could only store the latest entry. Multiple name-cache entries for a vnode occur when there are multiple hard-links to a file. If a cache-lookup operation found a matching entry that was not the latest one for a vnode, it would end up referencing a vnode that was already deallocated and result in bad memory access in the kernel.

4.3.2 Shadow name-cache

This design was to take care of the fact that multiple name-cache entries were possible for a vnode. When a name-cache entry for a vnode is being made, the name-cache address field in the vnode is checked to see if the vnode has an earlier entry. If so, the first and the second name-cache entries are linked by adding an extra address field to the name-cache entry structure. In this way, if a vnode had multiple name-cache entries, every entry will have a shadow entry link to the previous entry until the link reaches the first entry for the vnode. When the time comes to deallocate a vnode, the link from the first entry down to the last entry is followed and deleted.

This approach works very well on uni-processor systems, but has race problems in multi-processor systems. Several locks had to be introduced to follow the links to all the name-cache entries of a vnode. Each step required holding and releasing locks, and then looking back to see if anything had changed in the environment. The increase in the use of locks has a negative impact on the performance on multi-processor platforms. Despite these efforts, the implementation of this design was hard hit by race conditions.

4.3.3 Attaching vdealloc to the idle thread

Instead of having the vnode deallocation code attached to *vrele()* and *getnewvnode(),* it was considered a better idea to have a separate thread execute the code, thus allowing the task to be done when the system was relatively idle. So, the initial plan was to attach this code to the idle thread in the kernel. Further study showed that this was not feasible because the vnode deallocation code could block trying to get a vnode off the vnode free-list, as the free-list lock could be held by other processors in an SMP environment. A future enhancement of the design would be to use a separate kernel thread to execute this code.

5. Performance

Dynamic vnodes introduces a certain amount of overhead, but the design criteria was to have the performance very close to that of a similar system without dynamic vnode deallocation. It was also the goal to closely monitor the name-cache performance and to ensure that it is close to the case where name-cache entries are not invalidated based on their age.

Initially, a number of small exerciser programs were used to specifically test the vnode allocation/deallocation code. These programs forced the system to allocate a large number of vnodes and later release them to the free-list. These were originally designed to exercise the new code path but were also used to compare the performance of the systems.

Later, to find the overall performance of the system, standard benchmarks were used and the results were compared with the base-line reference provided by the version of the operating system before the introduction of vnode deallocation. All along, the name-cache performance was monitored by periodically extracting the name-cache statistics, called *nchstats*. A new counter called *bad_time_hits* was introduced to keep track of all matches that were thrown away as a result of the age of the entry being older than *name-cache-valid-time.*

In all cases, the cache-performance results showed that the discarded hits were less than 0.1% of the total good hits.

5.1 AIM Performance

5.1.1 Description of the test

AIM Suite III benchmark was developed by AIM Technology to test the total system performance of all major system components in a multitasking environment [AIM93]. Suite III attempts to simulate the load that a specified number of users would exert on a computer system by running a set of functional benchmarks intended to model a particular application. The set of functional benchmarks is scaled by the number of simulated users to arrive at the desired load. Besides file-system exercisers, these functional benchmarks also consist of memory, floating point, pipe, and CPU exercisers. This test was used to compare the performance of the kernel with and without vnode deallocation to determine if the overhead of having vnode deallocation was significant. The test results reported here are based on the *Standard Model,* which purports to describe a typical UNIX environment.

5.1.2 Setup for the test

The test was performed on a DEC 7000 AXP system running DEC OSF/1 Version 3.0 (Revision 344) with the following configuration:

- four DECchip 21064/182MHz
- 512MB RAM
- Prestoserve NVRAM
- six RZ28 (4GB) disks, two controllers

The tests were done with the network interfaces (Ethernet/FDDI) turned off and the test system was isolated from any external stimulus.

5.1.3 Results

The AIM Suite III Systems Throughput Comparison graph (Figure 2) shows the throughput comparison of two tests. The first test was conducted with vnode deallocation enabled (default). In the second test, vnode deallocation was turned off to avoid the overhead introduced by the deallocation code. It is evident by comparing the results of the two runs that there is hardly any drop in overall system performance when deallocation of vnodes is enabled. Note that the difference between the two curves is not always observed when the load exceeds 900 users. Therefore it cannot be attributed to the vnode deallocation code.

The name-cache usage statistics shows the performance of the name-cache. The statistics gathered after the AIM test from 1 to 1100 users are :

```
bad_time_hits          28
ncs_goodhits     51397595
ncs_neghits         75276
ncs_badhits        833068

vn_allocations      18299
vn_deallocations    10939
```

From the statistics, 28 good hits had to be rejected from a total of 51,397,595 due to the possibility of the corresponding vnodes being deallocated. This works out to be less than 0.000054% when the *name-cache-valid-time* value was set at 20 minutes. This shows that the two main design requirements were met in the implementation:

- overall performance remains unchanged after the implementation of dynamic vnodes.
- the name-cache performance drop is well under 0.1%, even though the name-cache entries older

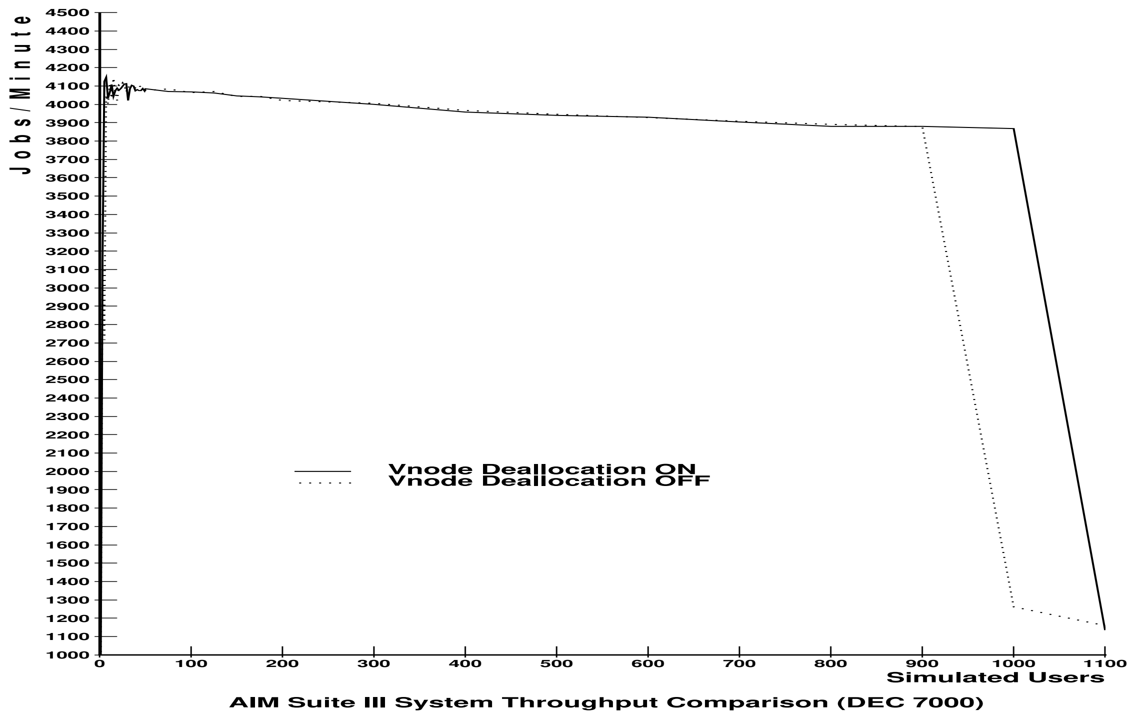

Figure 2: AIM Performance Graph

than *name-cache-valid-time* were being discarded.

5.2 UIW Performance

UIW is an acronym for a benchmark called ULTRIX Integrated Workload. This is used to determine the performance of a release of the operating system by measuring its response-time and throughput for varying load conditions.

5.2.1 Description of the test

UIW is a workload that attempts to simulate the activities of various members of a software development project on a time-sharing system. The workload is composed of a mix of commands and utilities that are file and CPU intensive (ls, cat, grep, make, cc, awk, etc.) [DEC-RP] running in a canned environment. The selection and frequency of execution of the processes are based on the observations of the activities of a software development unit in Bell Laboratories.

5.2.2 Setup for the test

The test was performed on a DEC 3000-500 AXP system running DEC OSF/1 Version 3.0 (Revision 344) with the following configuration:

- one DECchip 21064/150 MHz (uni-processor)
- 256 MB RAM
- four RZ28 (4GB) disks on a single controller

5.2.3 Results

From the results shown in Figure 3, the response time for the system that deallocates vnodes is close to that of the system that does not deallocate vnodes. In this graph, a system with lower response time is considered better. Figure 4 shows that throughput changes with increasing number of users are similar. Throughput is measured as the number of times the commands were executed in a duration of three hours, for a given number of simulated users. At the end of the run, the cache statistics were examined for losses due to vnode deallocation. From a total of 23,228,647 good hits, 9,425 hits were rejected because the name-cache entry was older than *name-cache-valid-time.* The loss was about 0.0405% of the good hits, which is well below the design requirement of 0.1%. Each UIW test takes about 48 hours to complete.

The results from the UIW tests show that the overall performance of DEC OSF/1 was not impacted by the addition of the vnode deallocation logic.

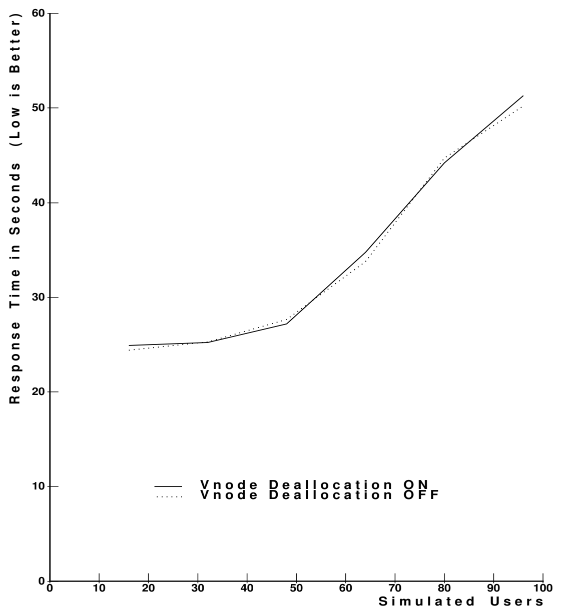

Figure 3: UIW Response Time Graph

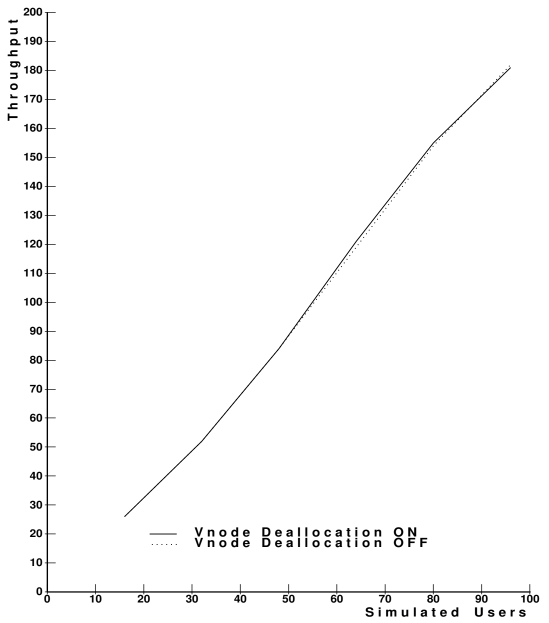

Figure 4: UIW Throughput Graph

6 Impact of the design

6.1 Impact on tools

As a result of the introduction of dynamic vnodes, all application and debugging programs that depended on the static vnode table locations had to be modified. They can no longer obtain the starting address of the vnode table, the size of the vnode table, and the offset to the start of a particular vnode. Instead they can only access the vnodes from the various lists by walking down the pointers in these lists. These lists will include the vnode free list and all of the mount lists. The mount lists are connected to each other in a circular list, starting with the *rootfs.* All kernel debugging tools that access vnodes were modified in DEC OSF/1 V3.0 to gather information on vnodes from the mount-lists and the free-lists.

6.2 Impact of tunable parameters

The tunable value of *name-cache-valid-time* can impact the performance of the *namecache.* If the value is set low, the deallocation will happen at a faster rate, while the number of bad cache hits will also increase. Higher values of *name-cache-valid-time* will slow the rate of deallocation, but will decrease the number of bad cache hits.

The tunable value of *vnode-age* will impact the size of the vnode free list. Large values will cache free vnodes for a longer time before the vnodes are re-cycled, but will increase the length of the free-list. If its value is set low, the vnodes will be re-cycled faster and allocation will be less aggressive, but the number of free vnodes cached will be less.

6.2.1 Selecting the default values

The design goal was to keep the name-cache performance loss due to cache invalidation under 0.1%. The value of *name-cache-valid-time* has a direct impact on name-cache performance. Starting with a value of 10 minutes, the value was incremented based on the feedback obtained from name-cache performance statistics recorded in the *nchstats* global kernel structure. A value of 20 minutes was chosen since it kept the losses under 0.1% for different types of loads. At the same time, it was important not to delay the deallocation process by choosing a very large value. A large value for *name-cache-valid-time* will weaken the deallocation process, leaving a large number of free vnodes in the system.

It is hard to come up with a default value for *vnode-age* that would be ideal for all types of work loads. In *vnode_stats* kernel structure*,* the *vn_vgetfree* field indicates the number of vnodes that were successfully taken off the vnode free list. If this value is too low, the value of *vnode-age* may be raised to increase the minimum time a vnode will remain on the free-list. A default value of 2 minutes is used. This value can be fine tuned on a running system, depending on the type of jobs that typically run on it. Its value will require further fine tuning in the future, depending on the type of machine and the type of load.

The value of *max-vnodes* is made sensitive to the amount of memory in the system. It's default value is calculated at boot time to the number of vnodes that can fit into 5% of memory. Although this value may seem high, only a fraction of that memory is used most of the time because it is just the upper bound for dynamic allocation of vnodes. Its value can be changed at any time on a running kernel.

The default value of *minimum-free-vnodes* is directly proportional to the configured value of the maximum number of users for the system. If necessary, this can also be changed on a running system. A kernel that has been configured for 32 users will typically have a default *minimum-free-vnodes* value of 468 vnodes.

7. Related Work

The ULTRIX Engineering Group at Digital developed a prototype kernel based on ULTRIX V2.0, where many of the static tables were replaced by linked lists [Rodriguez88]. However, vnode deallocation was not implemented.

The Open Software Foundation had done part of the work where vnodes get allocated, but did not implement the vnode deallocation code.

Similarly, IRIX Version 5.0 from SGI allocates vnodes and maintains a global free-vnode pool from which vnodes can get re-assigned.

4.4BSD-Lite (April 1994) also allocates vnodes, but does not deallocate them. According to sources from University of California, Berkeley, 4.4BSD-Lite is more advanced than the 4.4BSD-Encumbered, and vnode deallocation is not currently done in either of them.

8. Future work

Several areas can be worked on in the future to make the deallocation code even more optimal. Some of the areas are:

- adding a dedicated thread to do the deallocation of vnodes, instead of attaching it to *vrele()* and *getnewvnode()* code paths. This will reduce the extra overhead placed on the two calls.
- increasing lock optimizations and improving lock granularity of the locks used.
- doing name-cache entry invalidation based on name-cache activity time and the age of the vnode that was most recently deallocated, instead of on *name-cache-valid-time*.
- fine-tuning the default values of the tuning parameters to make deallocation even more optimal. Fine tuning is needed in the case of *vnode-age* to make vnode caching on the free-list more optimal.
- checking the amount of data associated with the vnode object before deallocating a vnode. It may be far less expensive to deallocate a vnode that has fewer pages of data associated with it.
- converting the static name-cache table with a dynamically managed list based on the number of vnodes in the system. The static name-cache table has been temporarily retained during the first phase of implementation, as the name-cache structures are a lot smaller than vnode structures.

9. Conclusions

Although the use of dynamic data structures is a well-known concept, the dynamic vnodes design shows new techniques to counter some of the issues due to the inherent nature of the existing code:

- data structures that contain references to a structure that has been deallocated can easily be purged by using timestamps. This eliminates the need for expensive searches to purge such data structures. This also eliminates the need to keep track of all possible references to a structure that can get deallocated.
- caching of free structures is more effective when the free structures are retained for a set duration, provided that there is enough memory. This balances the cache efficiency with memory consumption and prevents inadvertent cache flushes.
- tracking a marker element in a LRU list helps as an indicator of the traffic on the list. This information can be used to make the design sensitive to different patterns of demand.

These are general purpose techniques that can be used to resolve issues that are similar in nature to those encountered by dynamic vnodes.

Dynamic vnodes provide the system with all of the advantages of dynamic data structures. Because their allocation and deallocation is based on demand, they make the system more responsive to varying loads and increases its capacity to meet varying demand for vnodes. In this design, vnodes do not take up any more memory than required because none of the vnodes are pre-allocated. Dynamic vnode implementation in DEC OSF/1 also provides improved vnode caching on the free-list, thereby reducing disk access.

The results from the benchmarks show that the implementation of the deallocation design does not add any measurable overhead. The percentage of good name-cache hits lost due to aging is within tolerance limits.

9.1 Availability

Dynamic vnodes with allocation and deallocation are a part of DEC OSF/1 Version 3.0, which was released in August 1994.

10. Acknowledgments

I would like to thank all the members of the file-system group for their support. Special thanks to Stan Luke who motivated and supported me during the design and implementation. Paul Shaughnessy, Chet Juszczak, Eric Werme, and Jim Woodward helped me at various stages of this work. Diane Lebel provided invaluable technical reviews of various drafts of this paper. Bob Hapgood and Marty Lund helped with the performance measurements. Ken Hall improved the readability of this paper. Rich Draves at Microsoft, my USENIX shepherd for this paper, provided valuable suggestions and detailed reviews beyond the call of duty. Cheryl Wiecek, Rich Cascio, and Andy Kegel supported me in publishing this work.

Bibliography

[AIM93] "UNIX System Price Performance Guide", *Summer 1993, AIM Technology, Santa Clara, CA.*

[Bach84] M. J. Bach, S. J. Buroff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Labs Technical Journal*, Vol 63, October 1984, 1733- 1749.

[Bach86] M. J.Bach,"The Design of the UNIX Operating System", *Prentice-Hall, Inc.,* 1986, Chapter 4.

[DEC-RP] "DEC OSF/1 Reference Pages, Version 3.0." *Digital Equipment Corporation, Maynard, MA.*

[DEC-ST] "DEC OSF/1 System Tuning and Performance Management, Version 3.0." *Digital Equipment Corporation, Maynard, MA.,* 1994, Chapter 3.

[Kleinman86] S. R. Kleinman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the USENIX Summer '86 Conference*, 238 - 247.

[Langerman90] A. Langerman, J. Boykin, S. LoVerso, S. Mangalat, "A Highly-Parallelized Mach-based Vnode Filesystem", *Proceedings of the USENIX Winter '90 Conference,* 297 - 311

[LoVerso91] S. LoVerso, N. Paciorek, A. Langerman, " OSF/1 UNIX Filesystem", *Proceedings of the USENIX Winter '91 Conference,* Jan 1991, 207 - 218.

[McKusick88] M.K. McKusick, and M. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3 BSD UNIX Kernel", *Proceedings of the USENIX Summer Conference,* June 1988, 295 - 303.

[OSF93] The Open Software Foundation,"Design of the OSF/1 Operating System - Release 1.2", *Prentice Hall, Inc.,* 1993, Chapter 11.

[Rodriguez88] R. Rodriguez, M. Koehler, L. Palmer, R. Palmer, "A Dynamic UNIX Operating system", *Proceedings of the USENIX Summer Conference,* June 1988, 305 - 319

Author Information

Aju John is a software engineer in the UNIX Engineering Group at Digital, where he works in the file systems area of DEC OSF/1. He has worked at Digital since 1991. Aju got his M.S. in C.S. at the Worcester Polytechnic Institute in Worcester, Massachusetts in 1991. Reach him electronically at aju@zk3.dec.com or via U.S. Mail at Digital Equipment Corporation, ZK03-3/U14, 110 Spit Brook Rd, Nashua, NH 03062-2698.