

# MemFS v2 – A Memory-based File System on HP-UX 11i v2

Technical white paper



Introduction .....	2
Why a Memory-based File System? .....	2
The HP-UX Memory-based File System Design .....	3
MemFS user process backing store .....	4
Mount options .....	4
Performance.....	5
Postmark Benchmark .....	5
Connectathon Tests .....	7
SDET Benchmark.....	8
MemFS mount/umount performance.....	9
Configuration Guidelines.....	9
Recommended Use .....	11
Limitations and Future Work.....	11
Summary .....	11

## Introduction

A Memory-based File System (MemFS) is a file system that resides in memory. It does not normally write data out to stable storage. A MemFS is created from a mount operation, and ceases to exist when it is un-mounted. The purpose of such a file system is to provide fast access for temporary files that do not need to be kept for an indeterminate time. Because it does not normally have to do I/O to stable storage, the MemFS is able to provide extremely high throughput.

Keeping data in memory comes at a cost. It consumes system physical memory. A system or application that runs out of available memory at a critical time can cause irreparable loss to the user. That is why most of the Virtual Memory (VM) management systems implement a paging policy, wherein less frequently used memory pages are paged out to a swap device. This policy has been extended to the MemFS. Under memory pressure the VM system can de-allocate MemFS pages and reassign them where needed.

MemFS requires a user process to be associated with each of its instances. The memory allocated in the process address space of this user process is used to store the data from the reallocated pages. These user memory pages can be paged out to the system swap device whenever there is system memory pressure.

This document discusses how MemFS in HP-UX 11i v2 is designed and it differs from a RAMdisk. It will also cover the performance implications and guidelines to configure MemFS.

## Why a Memory-based File System?

A memory-based file system is typically used as storage for temporary files. By keeping as much data as possible in memory it avoids having to perform disk I/O and the associated overhead. Traditional file systems manage two types of data. One is the file content (called data), which an application accesses through the read, write and mmap mechanisms. The other, called metadata is the information related to file attributes and the file system structure.

Most file systems implement a buffering mechanism for data. That is, a copy of the disk data blocks is maintained in memory (buffer cache or page cache, based on the implementation). This buffer is used to service the reads from a disk, and to implement a “delayed write” mechanism. Using a buffer helps the file systems avoid disk I/O to some extent and improve the file system performance. Structural and attribute changes are buffered to a lesser extent, and will usually cause disk I/O to be done.

A memory-based file system goes a bit further to avoid disk I/O. Metadata is kept in memory, so the overhead of writing structural changes to the disk is totally avoided. MemFS data blocks are treated mostly at par with other file system blocks and can be swapped out of the buffer cache. There is a difference, though. Disk based file systems try to keep the disk copy up to date with the buffer copy. So, “dirty” buffers (those that have been modified) are periodically written back to the disk. This is important since these file systems are expected to preserve data across system shutdown and reboot. A memory-based file system has no such requirement, so data is written to the disk only if that buffer is being paged out.

Before memory-based file systems became popular, the concept of a “RAM disk” existed. A RAM disk reserves a range of memory and makes it available through a block device interface using a pseudo driver. The block device interface permits a file system to be created on it, in effect providing a memory-based file system. However, RAM disks have certain drawbacks. Since memory is reserved at the time of the disk creation, it is locked from shared system use, whether actually in use or not. Most RAM disks do not support paging of their memory, resulting in very poor system response in cases where the system is running low on free physical memory.

Since the system sees a RAM disk as a device rather than a file system, any access to a file stored on it results in a second copy of the data being kept in the file system buffer. Some of the implementations of a RAM disk may provide an improved interface and some of these shortcomings may not be present.

As attractive as the performance aspects sound, a memory-based file system only works efficiently under certain circumstances. The biggest disadvantage is that it does not preserve data across mounts. It cannot be used to replace file systems that store persistent data. Since it uses memory which is a shared resource, excessive use can adversely affect other memory consumers. A typical use of a memory file system is for storing temporary files that are created, accessed frequently in a short span of time and then deleted. The advantage comes from not having to update large amounts of metadata on file creation – growing the file – and then deletion and cleanup.

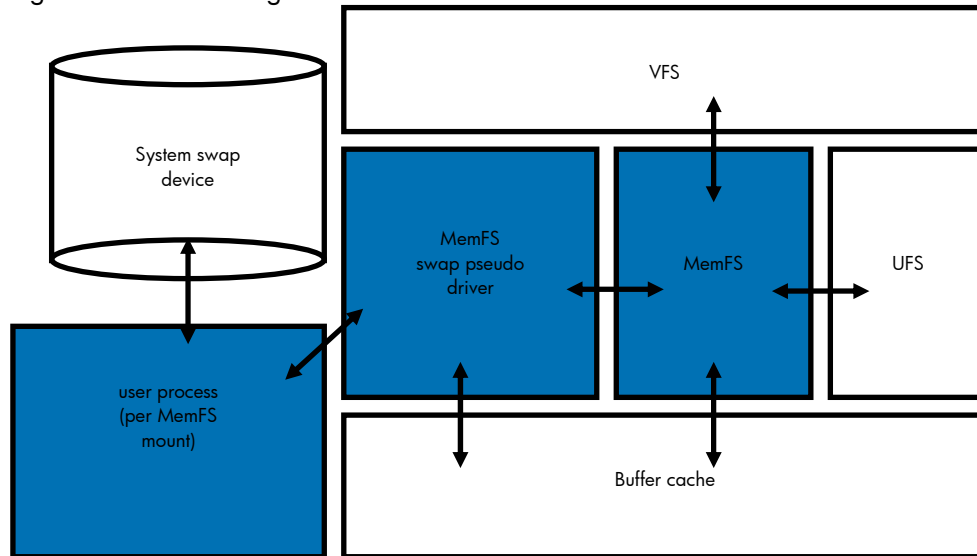
The write-to-swap feature of memory-based file systems efficiency has direct correlation with the swap activity. As the swap activity increases, the file system performance decreases.

## The HP-UX Memory-based File System Design

The memory file-based system which is available as a configurable product in HP-UX 11iv2 is based on the buffer cache approach.

HP-UX uses HFS<sup>1</sup> as the base for MemFS<sup>2</sup>. HFS provides a well-designed and tested file system implementation. It has all the features needed for a basic file system. Some of the more advanced file system features (e.g. journaling, un-buffered I/O) that some of the contemporary file systems provide are neither required nor relevant in the context of a memory file system.

Figure 1. MemFS Design



A MemFS file-system is created from a `mount(1M)` command. Initially the metadata for the file system is created in memory, and the necessary VFS<sup>3</sup> and inode structures are initialized. A file system has the concept of disk layout, which refers to how information such as the superblock, inodes and the data blocks are organized on the disk. In a MemFS this disk information is simulated, and kept in

<sup>1</sup> HFS – Hierarchical File System, the file system traditionally used by Unix based Operating Systems

<sup>2</sup> The term MemFS is used to refer to the HP-UX implementation of a Memory File System.

<sup>3</sup> VFS- Virtual File System – an abstraction layer on top of a more concrete file system.

memory. The mount results in a combination of an mkfs(1M) and a mount(1M) without the disk operations.

Subsequent to a mount, the MemFS can be used just like any other file system, with the same commands and system calls. From an application or user perspective there should be nothing that differentiates it from any other file system, except a better response time.

Internally, MemFS behaves differently from other file systems. The syncer (1M) <sup>4</sup>daemon, which periodically flushes “dirty” file system pages to disk, does not have an effect on MemFS buffers, as MemFS buffers are never added to the syncer dirty list - which is processed periodically for flushing. Periodically the VM paging algorithm reallocates the less used pages from the buffer cache and user address spaces. MemFS data pages, too, can be recycled, and written to the user memory allocated in the user address space of a process associated with each mount instance. When accessed, the data is retrieved and copied to the buffer cache.

A umount (1M) on the MemFS removes the file system and all its data. As per the specifications, data cannot be recovered once a umount (1M) happens.

## MemFS user process backing store

MemFS requires a user process to be created for every MemFS mount instance. The address space of the user process is used to store less frequently used MemFS data pages that have been moved out of the buffer cache. The user process continues to exist as a daemon, sleeping in the kernel, as long as the MemFS instance is mounted.

The decision to use a user process as the backing store was driven by the fact that the HP-UX 11iv2 kernel does not have support for page-able kernel memory.

HP-UX 11iv2 file systems use the buffer cache approach for managing the data buffers. HP-UX 11iv3 has a unified file cache (UFC) model.

MemFS stores all data and meta-data in the buffer cache. MemFS buffers are treated differently from other buffers in that they are not normally chosen for replacement through LRU policies. Since MemFS file data is stored in the buffer cache, rather than cached like other file systems, performance will be improved. However, when the number of MemFS buffer pages exceeds the number computed by the tunable memfs\_bufcache\_swappct, data from least recently used MemFS pages will be copied to the user address space of the associated user process. These MemFS buffers are then released back to the buffer cache pool. Pages in the user page-able memory can be chosen by the paging daemon to be paged out to the system swap device and brought back to the buffer cache when they are accessed.

A pseudo driver called ‘memfsswap’ acts as a pseudo disk and maintains a dynamic mapping of MemFS memory pages and their virtual disk location on the user process.

## Mount options

Since the behavior of MemFS is quite different from other file systems, the mount options are different.

The syntax for mounting a MemFS is:

```
/sbin/mount -F memfs [-eQV] [-o specific options] directory
```

Note that a block special filename is not required for MemFS.

---

<sup>4</sup> The terms used here refer to those used with the HP-UX. Other implementations may use a different terminology.

Also, note that the `-r` option is not supported. Since a read-only MemFS will always remain empty, this option, applied to a MemFS, serves no purpose. Other HFS options such as `behind`, `delayed`, `fs_async`, `no_fs_async` are not supported since they do not have any relevance in a Memory-based File System context. MemFS does not support quotas.

The file-system specific option that MemFS supports is:  
`size` – Specifies the maximum size to which this MemFS instance is allowed to grow.  
A MemFS can also be automatically created at boot time by making an entry in `fstab` (4)  
`memfs directory memfs size=<size> 0 0 #comment`

Since there is no backing device directly associated with MemFS, `special` is ignored, and it is recommended that “`memfs`” is used as special field.

The `umount` (`1m`) syntax is:  
`/usr/sbin/umount [ -v ] directory`  
where `directory` is the mount point for the MemFS instance

## Performance

A MemFS is expected to perform much better than a normal disk file system for smaller file and file system sizes. As the memory occupied by MemFS increases to a level where swapping starts happening, the performance may begin to taper off. If the swapping becomes excessive, performance will decrease.

MemFS performance varies depending on usage and machine configurations. The benchmark tests chosen to characterize the performance of MemFS are: Postmark, Connectathon basic tests and SDET.

## Postmark Benchmark

Network Appliance’s Postmark is an appropriate benchmark to use for MemFS as it is an industry-standard benchmark for small file and metadata-intensive workloads. It was designed to emulate Internet applications such as e-mail, netnews, and e-commerce.

Postmark works by creating a configurable sized pool of random text files ranging in size between configurable high and low bounds. These files are continually modified. The building of the file pool allows for the production of statistics on small file creation performance. Transactions are then performed on the pool. Each transaction consists of a pair of create/delete or read/append operations. The use of randomization and the ability to parameterize the proportion of create/delete to read/append operations are used to overcome the effects of caching in various parts of the system.

### Test Configuration and Methodology

The system under test consisted of a HP rx7620 server comprising of 2 x 1600MHz Itanium CPUs, 32GB RAM. Postmark v1.5 was used.

The following graphs show the effects of applying increasing transaction loads to the various file systems. Larger figures indicate higher performance.

The Postmark benchmark used the following parameters:

```
set size          10000 15000      (Sets the low and high bounds of files)
set number        10000           (Sets the number of simultaneous files)
set transactions  RUN_LOAD        (Sets the number of transactions)
set report verbose
set subdirectories 0 100
run
```

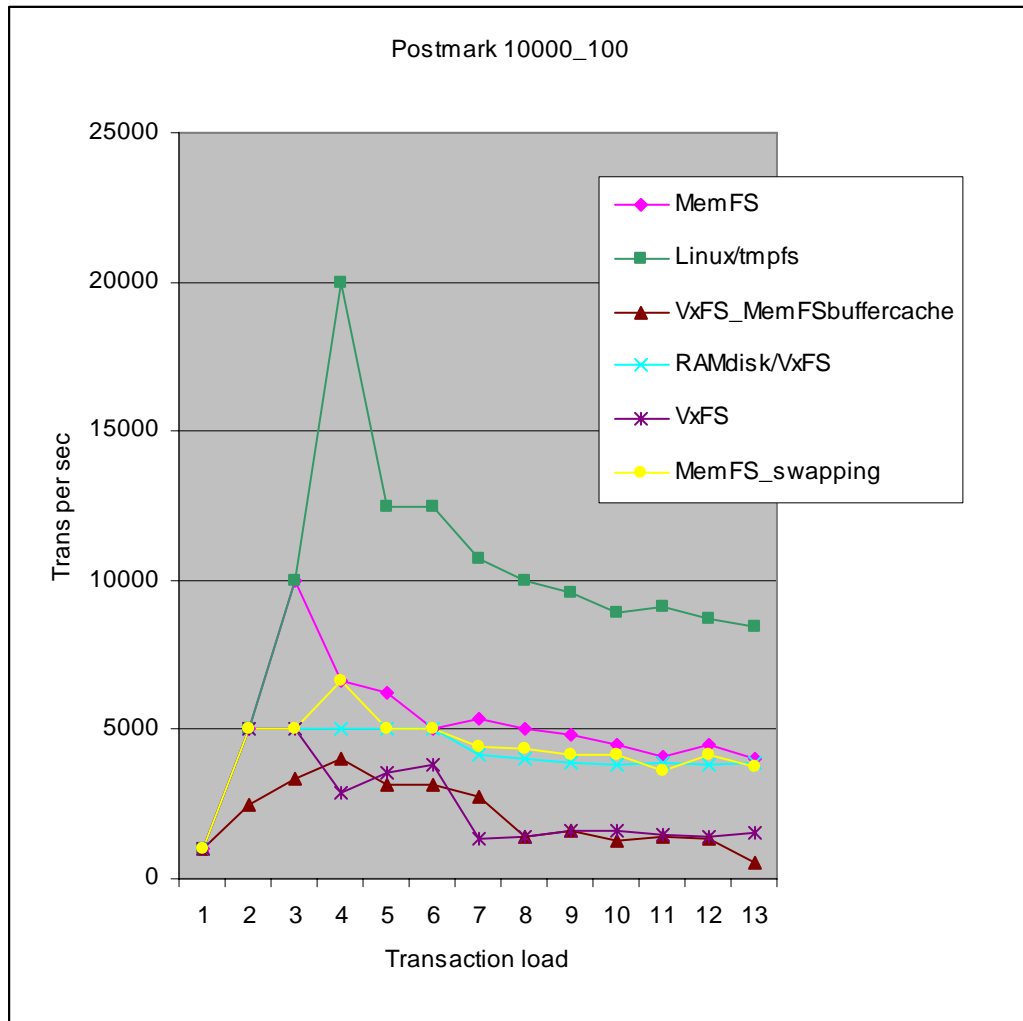
```
quit
RUN_LOAD={1000, 10000, 25000, 75000, 125000, 300000, 500000}
```

The filesystems under test were:

- tmpfs on Red Hat Linux 2.4.21-4
- MemFS on HP-UX 11iv2 of size=2GB. dbc\_max\_pct was set to 50% (4GB)
- RAMdisk on HP-UX 11iv2: Basefile system: VxFS 3.5 using the delaylog mount option.
- VxFS 3.5 on HP-UX 11iv2 was mounted with the highest caching options: tmplog, mincache=tmpcache, convosync=delay

The following graph shows Postmark results operating on 10,000 simultaneous files spread across 100 subdirectories. Linux's tmpfs shows the best performance, which can be attributed to a lightweight filesystem design and a large system page size of 16k. However, it can be seen that when the transaction load increases, tmpfs will start swapping and the performance can drop. MemFS (even while in swapping mode) performs better than VxFS and RAMdisk(VxFS). Also shown is VxFS performance when the buffer cache is filled with MemFS buffers, resulting in marginal performance degradation.

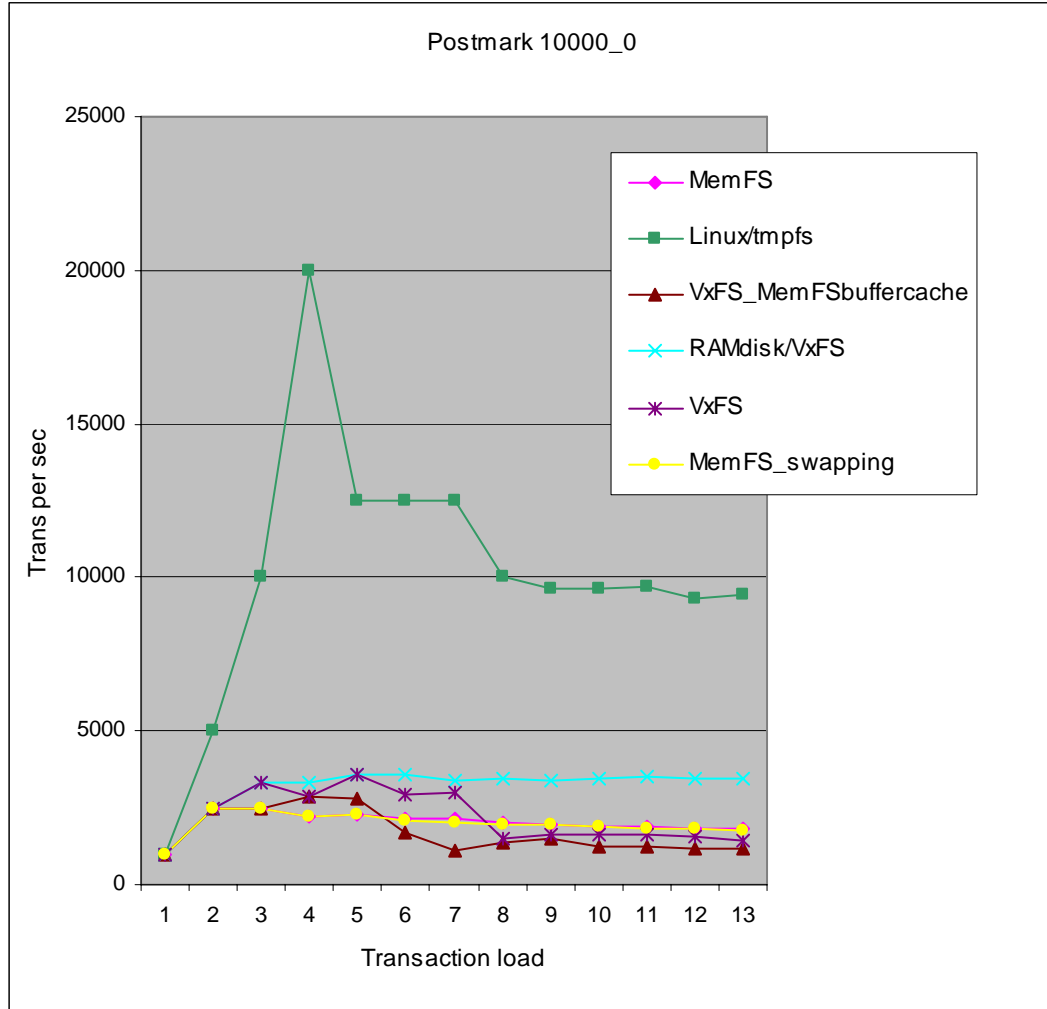
Figure 2: Postmark benchmark results for 10,000 simultaneous files and 100 subdirectories



The following graph shows Postmark results operating on 10,000 simultaneous files created all under one directory. MemFS in such instances does not perform too well. This can be attributed to linear

directory lookups in HFS code that MemFS uses. tmpfs is not affected by the number of directory entries.

Figure 3: Postmark benchmark results for 10,000 files and no sub-directories



The tests indicate that significant performance gains can be achieved in HP-UX 11iv2 by using MemFS, as long as the number of files of a directory is not too large.

## Connectathon Tests

The Connectathon tests are a collection of micro-benchmarks used to benchmark NFS. The basic tests are used to verify basic operations of "NFS" file system implementations. The following tests were used:

create	Create a directory tree 6 levels deep
remove	Remove the directory tree from create
lookup	stat a directory 500 times
setattr	chmod and stat 10 files 1000 times each
read/write	write and close a 1MB file 10 times, the read the same file 10 times
readdir	read 20500 files in a directory 200 times
link/rename	rename, link and unlink 10 files 200 times
symlink	create and read 400 symlinks on 10 files
staffs	stat the mount point 1500 times

The system under test consisted of a HP rx5670 server comprising of 2 x 900MHz Itanium CPUs, 8GB RAM.

Figure 4: Connectathon benchmark results

	Hfs	VxFS		Ramdisk	MemFS (sec)	Linux ext3fs (sec)	Linux tmpfs (sec)
	Default (sec)	Log (sec)	Delaylog (sec)	VxFS (sec)			
Create	2501.46	699.41	23.48	6.44	4.91	197.51	0.82
Remove	989.77	677.42	8.54	5.38	3.82	26.88	0.91
Lookup across mounts	0.0	0.0	0.0	0.3	0.0	0.0	0.0
Setattr	0.0	6.0	0.1	0.1	0.0	0.0	0.0
Write	0.23	0.54	0.52	0.2	0.3	0.4	0.0
Read	0.1	0.1	0.1	0.1	0.1	0.0	0.0
Readdir	1.24	1.2	0.1	0.1	0.1	0.2	0.1
Link and Rename	3.71	1.8	0.6	0.1	0.0	0.0	0.0
Symlink and Readlink	3.73	2.51	0.1	0.1	0.1	0.0	0.0
Stafs	0.0	0.0	0.0	0.0	0.0	0.22	0.0

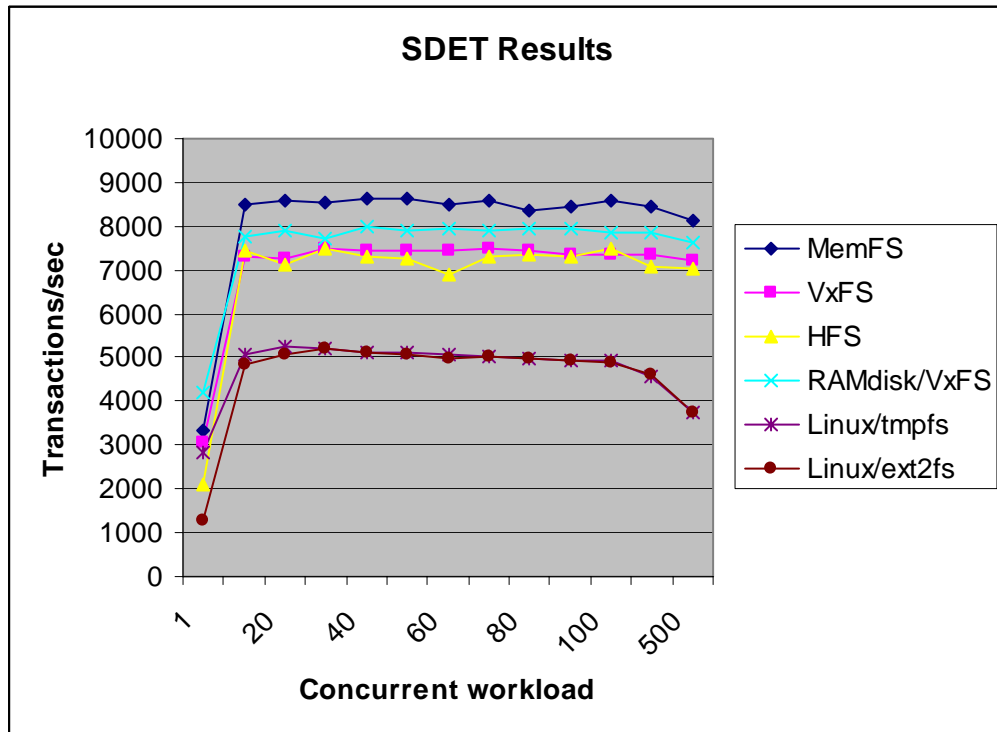
The Connectathon basic test results indicate that MemFS performs slightly better than RAMdisk/VxFS, and far better than disk based filesystems, but not as good as Linux tmpfs.

## SDET Benchmark

The Software Development Environment Throughput (SDET) benchmark is a highly scalable, multi-process benchmark which exercises most of the kernel (except networking). The benchmark was designed to apply a representative model workload to a system at increasing levels of concurrency in order to generate a graph of throughput vs. offered load. SDET, is however, a system benchmark, and therefore may not give an accurate picture of filesystem performance only.

The system under test consisted of a HP rx5670 server comprising of 2 x 900MHz Itanium CPUs, 8GB RAM. The same test system was installed with Red Hat Linux 2.4.21-4 and the SDET benchmark was run on tmpfs and ext2fs.

Figure 5: SDET Benchmark results



The following conclusions can be drawn:

- MemFS performs better on the SDET benchmark compared to disk based filesystems and RAMdisk
- MemFS performs better on the SDET benchmark compared to Linux's tmpfs on the same test machine

## MemFS mount/umount performance

Since MemFS needs to create and initialize all meta-data structures during a mount operation, there will be a noticeable amount of time taken to create large MemFS instances. User process memory whose size is equal to the size of the file system needs to be pre-allocated at mount time. In addition, MemFS meta-data structures are stored in the buffer cache and the buffer cache resources are not released until the MemFS instance is unmounted. Therefore, it is recommended that the size of the MemFS instance is chosen appropriately as required during mount.

Similarly, the umount operation may take some time to complete for a large MemFS instance, as MemFS needs to de-allocate all related structures and free up resources by traversing through the entire buffer cache looking for buffers that need to be invalidated.

## Configuration Guidelines

A MemFS instance requires allocation of a memory area to be associated with it, whose size is equal to the size of the MemFS instance. The mount command will fail if it cannot allocate enough memory. Sufficient disk space must be configured to the system swap device for mount to succeed. If the MemFS file system size to be created is large (greater than 2 GB), the tunable `maxdsiz_64bit` must be tuned appropriately. This memory area is allocated in the user address space of a process that is forked off the mount command for every mount.

The maximum number of MemFS file system instances supported in HP-UX 11iv2 is 64.

A MemFS instance can be created either through a mount(1) command, or automatically through an entry in the /etc/fstab file. Creation of a MemFS instance through the mount(2) system call is not supported.

The syntax for mounting a MemFS instance using the mount(1) command is:

```
/usr/sbin/mount -F memfs [-eQV] [-o size=<size>] directory
```

for example, to create a MemFS instance of size 100MB and mounting it on /memfs:

```
/usr/sbin/mount -F memfs -o size=100MB /memfs
```

The optional size argument can be used to specify the size of a MemFS instance. Append to size, kb or KB to indicate the value is in kilobytes, mb or MB to indicate megabytes, or gb or GB to indicate gigabytes. If the size option is not specified, the default size is computed as the percentage of maximum buffer cache size that can be occupied by MemFS, as specified by the memfs\_bufcache\_swappct tunable.

Tuning the MemFS buffer cache size is very important because MemFS stores data and metadata there.

The amount of memory in the buffer cache used for MemFS can be tuned using the following dynamic tunables on HP-UX 11iv2. Both tunables are dynamic tunables and can be modified without a system reboot.

---

### **memfs\_bufcache\_swappct**

is the percentage of buffer cache at which MemFS buffers start swapping. The default value is 50. The memfs\_bufcache\_swappct tunable can be as low as 0 and can be as large as 100. If memfs\_bufcache\_swappct is set to 0, MemFS data buffers would start swapping immediately to the user process. If memfs\_bufcache\_swappct is set to 100, MemFS can occupy the entire buffer cache before it starts swapping. The larger memfs\_bufcache\_swappct is, the greater the amount of buffer cache that will be reserved for MemFS pages and the better the performance of MemFS. But, this may adversely impact the performance of other filesystems that use the buffer cache.

### **memfs\_bufcache\_metapct**

is the maximum percentage of buffer cache which can accommodate MemFS metadata of all MemFS instances. The default value is 10. The memfs\_bufcache\_metapct tunable cannot be less than 10 and cannot be greater than 50. Tuning memfs\_bufcache\_metapct to a value as high as 50 may be necessary if the system primarily uses large MemFS filesystems. The larger memfs\_bufcache\_metapct is, the more of the buffer cache will be reserved for MemFS metadata pages and hence will allow creation of more and larger MemFS filesystem instances, but impact the performance of other filesystems that use the buffer cache. Since MemFS metadata pages can never be swapped out to the user process, exercise caution when tuning memfs\_bufcache\_metapct as this may severely impact normal performance of other filesystems.

---

MemFS' behavior is similar to HFS and internally uses the same resources as the HFS filesystem like mount tables and inode cache. Therefore, tunables like ninode (which specify the maximum number of HFS file system open inodes that can be in memory) need to be modified appropriately.

## Recommended Use

Applications that will benefit most from MemFS are those that perform mostly meta-data intensive operations like creates and deletes of small files and directories. For example: compilers, editors and sorting applications. MemFS never writes out meta-data for files, so there is always a performance gain for directory and file manipulation. Applications can also interact with MemFS files using the mmap interface. Using the mmap interface instead of the read/write interface may be required for some applications.

Some other usage guidelines are:

- Use MemFS only for temporary files-
- Distribute the files evenly across a larger number of directories. If all files are accessed simultaneously, limit the number of files to about 100 per directory or so.

## Limitations and Future Work

MemFS uses the Hierarchical File System (HFS) as its base, and, as a result, it inherits several HFS functions that increase usage overhead. This includes the overhead of a disk based filesystem structure, static allocation and pre-initialization of inodes. Also, lack of kernel page-able memory in the HP-UX kernel has created the need to have user memory to back up MemFS pages, so that they can be transparently swapped to the system swap device.

These limitations will be addressed in a future MemFS version. Addressing these limitations will ensure that memory resources are managed more efficiently.

## Summary

MemFS shows tremendous performance advantages in comparison to memory-based file systems and also provides significant design advantages over traditional RAM disk style file systems. MemFS provides additional file system space, and supports UNIX file semantics while remaining fully compatible with other filesystem types.

Since MemFS is integrated with the virtual memory sub-system, it takes advantage of the kernel's resource management policies, and pages used by MemFS can be reclaimed back by the system.

© 2007 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Linux is a U.S. registered trademark of Linus Torvalds. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group.

4AA0-xxxxENW, August 2007

