Improving the efficiency of UNIX file buffer caches

Andrew Braunstein, Mark Riley, and John Wilkes
Hewlett-Packard Company

This paper reports on the effects of using hardware virtual memory assists in managing file buffer caches in UNIX. A controlled experimental environment was constructed from two systems whose only difference was that one of them (XMF) used the virtual memory hardware to assist file buffer cache search and retrieval. An extensive series of performance characterizations was used to study the effects of varying the buffer cache size (from 3 Megabytes to 70 MB); I/O transfer sizes (from 4 bytes to 64 KB); cache-resident and non-cache-resident data; READ*s and* WRITE*s; and a range of application programs.*

The results: small READ/WRITE *transfers from the cache (≤1 KB) were 50% faster under XMF, while larger transfers (≥8 KB) were 20% faster. Retrieving data from disk, the XMF improvement was 25% and 10% respectively, although* OPEN/CLOSE *system calls took slightly longer in XMF. Some individual programs ran as much as 40% faster on XMF, while an application benchmark suite showed a 7–15% improvement in overall execution time. Perhaps surprisingly, XMF had fewer translation lookaside buffer misses.*

1 Introduction

As processor speeds increase faster than disk access times decline, larger file buffer caches are being used to compensate for the performance differences between main memory (primary storage) and file backing store (secondary storage). One effect is that the cost of access to the file buffer cache will have a greater impact on overall system performance. The work described here explores one mechanism for minimizing these access costs: mapping files into a processor's virtual address space, and then using virtual memory hardware assists (such as a Translation Lookaside Buffer, or TLB) to speed cache lookups.

The research was conducted by comparing two otherwise-identical designs of a UNIX[1] file buffer cache: HP-UX and XMF. HP-UX, Hewlett-Packard's version of UNIX, is internally a 4.2BSD[2] derivative with a file system that utilizes a *software-managed* file buffer cache. XMF, the *Xylophone Mapped File* system, was derived from HP-UX by adding code to use the virtual memory hardware to map files into the kernel's virtual address space, thereby generating a *hardware-assisted* file buffer cache, which obviated referring to a file's disk map during accesses to data in the cache. To allow direct comparison of the old and new buffer management schemes, XMF preserved the HP-UX I/O semantics exactly (e.g. it used the traditional UNIX READ/WRITE system-call interface).

This paper presents some of the results of the research. Section 2 discusses related work in this field; section 3 briefly outlines the XMF design (more details can be found in [Braunstein89]); section 4 reports the experimental setup used; section 5 the results obtained from taking "micro" measurements; section 6 the results of running a set of "macro" benchmarks; and section 7 offers an analysis. Section 8 draws some conclusions from the work. Appendix I provides more information on the changes we made to HP-UX, and Appendix II contains some statistical data.

2 Related work

There have been many studies on caching within file systems (e.g. [Swinehart79, Satyanarayanan85,

This paper originally appeared in: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, The Wigwam, Litchfield Park, Arizona, and was published as *Operating Systems Review* **23**(5):71–82, December 1989.
Note: this paper has been slightly reformatted here from the original version, and the graphic of Figure 22 has been regenerated from the original data. No text has been changed.

[1]UNIX is a registered trademark of AT&T Bell Laboratories.
[2]The 4.2BSD software is release 4.2 of the Berkeley Standard Distribution of UNIX, distributed by the University of California at Berkeley.

Schroeder85, Cheriton87, Gifford88]), but these investigations were not primarily concerned with quantitative comparisons of file buffer cache access methods, or the impact of large physical memories on their performance. Giving more memory to a file buffer cache can boost system performance, but it can also have negative effects: [Feder84] reported that increasing the buffer cache size (and hence the number of cache blocks) sometimes resulted in lower overall performance, because the cost of finding data in the cache increased as the cache got larger.

Measurements reported in [Ousterhout85] showed that using a delayed write strategy a 4 MB UNIX file buffer cache had an 86% hit rate, and a 16 MB cache had a 90% hit rate. Simulations of a 128 MB file buffer cache showed that in steady state only 30–40 MB were used over a three day period. A similar study at Hewlett-Packard showed that a 60 MB cache would have a 99% hit rate [Holt87]. Thus, it would seem that in a computer system with a very large file buffer cache, say 60 MB, efficient cache *replacement* strategies would be less important than *locating and retrieving* cached file blocks quickly. One way of doing the latter is to map files into the virtual address space of the processor, and use the virtual memory hardware to assist in their access. Such mapped file systems were originally known as *single level stores* because they treated primary memory and backing store as a single entity.

The first use of a single level store was the Atlas system [Kilburn62]. The next major system to use the concept was Multics [Bensoussan72, Organick72]. Since then, it has been used in several others (e.g. [Henry78, Redell80, Leach83, Fitzgerald86, OQuin86, Busch87, Simpson87, Chang88, Nelson88, Ousterhout88, Tevanian87, Tevanian88]). The work reported here differs from these other endeavors by concentrating on isolating and measuring the effects of just one design decision, rather than including mapped-file techniques in a system that also altered many other design parameters.

XMF speedup		
	cache	disk
Small ($\leq$1 KB) transfers	50%	25%
Large ($\geq$8 KB) transfers	20%	10%
8 KB-READ bandwidth	15%	–
Individual applications	up to 30%	
Benchmark suite	7–15%	

XMF slowdown	
OPEN/CREATE & CLOSE	$< 9\%$
VM daemon overheads	$< 0.3\%$

Table 1: Summary of results.

3 XMF

Since blocks in the file buffer cache are aligned on physical memory page boundaries, the software buffer cache lookup scheme can be replaced by the hardware's virtual address translation mechanism. XMF uses this technique to find data in its file buffer cache. We hypothesized that this would provide a substantial performance benefit, particularly for small transfer sizes where we expected that the cache lookup overhead would dominate the cost of copying the data. Also, we expected that software lookup schemes would be more costly at large cache sizes.

XMF was explicitly designed for comparing the two file buffer cache lookup strategies. It is built into an existing HP-UX kernel, with the changes deliberately limited to those areas of the operating system that manipulate the file buffer cache. For example, XMF's internal virtual memory objects are not exposed to user applications, and XMF offers exactly the same system calls and system call semantics as HP-UX. The total amount of physical memory available for the file buffer cache is the same in the two designs. In XMF, a portion of its cache (typically 0.5 MB) is statically allocated to inodes and indirect blocks. The two systems use identical file system structures on disk; they can both run off the same disk pack. The XMF page-replacement daemon is separate from the HP-UX one, to minimize the interactions between them, to limit the scope of changes to the HP-UX kernel, and to allow the two daemons to be optimized for different use patterns. The end result is a pair of systems that differ in only one important aspect: how they access and manage their file buffer caches.

3.1 XMF implementation

XMF runs on Hewlett-Packard's Precision Architecture (HP-PA [Mahon86]) machines. HP-PA is a RISC-like pipelined LOAD/STORE architecture with 32 general purpose registers, a single, global virtual address space up to 2^{64} bytes in size segmented into 2^{32} byte *spaces*, 2 KB physical pages, an inverted page table, and software-managed TLBs. The HP-UX implementation for HP-PA is further described in [Clegg86].

XMF code sits between the HP-UX file system-call interfaces and the low-level I/O subsystem (Figure 1). The old HP-UX code is retained to manage non-disk files (e.g. block special devices, pipes, network special files).

Disk files in both HP-UX and XMF are described by *inodes*, which provide a mapping between logical file block numbers and physical disk block addresses (Figure 2). Since files can grow up to 2^{32} bytes long, a tree of inode indirect blocks may be needed to hold all the disk addresses. In typical HP-UX file systems, indirect blocks start to be used once a file grows beyond about

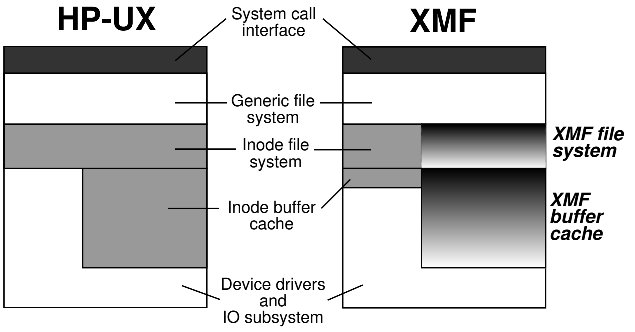

Figure 1: System structures of HP-UX and XMF.

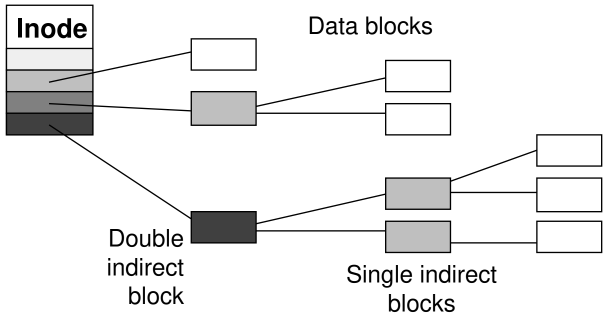

Figure 2: Inodes and indirect blocks.

90 KB in size. Both XMF and HP-UX cache recently-used blocks of files in a *file buffer cache*, an array of page-aligned physical memory slots.

The primary XMF design goal was to minimize the READ/WRITE instruction paths. Figure 3 shows the relative complexity of the HP-UX and XMF call graphs for a READ system call.

In XMF, a Virtual Storage Object (VSO) describes a virtual memory object—a range of addresses that represents a single logical object and its access rights (Figure 4). At file OPEN time, a VSO is created, and associated with its disk storage by constructing a Mapping Table Entry (MTE). Each MTE maps a contiguous virtual address range onto (part of) a single Active Storage Object (ASO), an abstraction for the underlying storage object that corresponds one-to-one with an HP-UX inode. (For these experiments, each MTE fully mapped a single ASO, and each file was mapped into a separate 2^{32} byte virtual space.) An MTE can also refine the access rights of its enclosing VSO. The MTE holds the information needed to construct the architected HP-PA data structures that describe the virtual to physical address translation: an inverted page table (PDIR) and its TLB entries.

When the last open file descriptor referring to a file is removed, the file's VSO is deleted, and the MTE slots are marked as free and added to a Least Recently Used (LRU) chain. If, as often happens, the file is re-opened

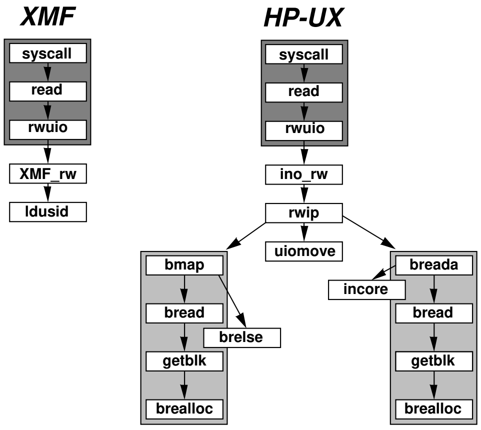

Figure 3: READ call graphs for XMF (left) and HP-UX (right).

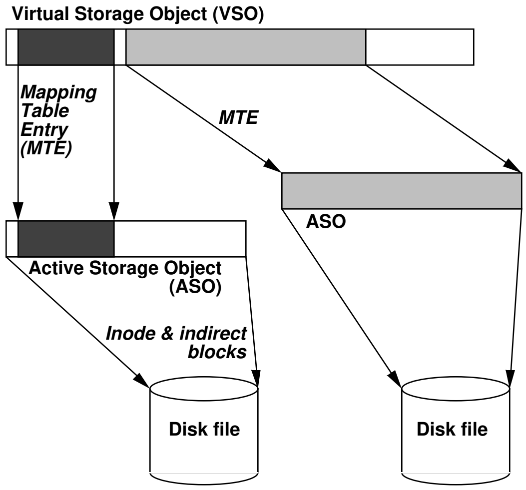

Figure 4: Virtual object management and physical memory management structures in XMF.

quickly, the MTE mappings can be reused; if not, any dirty buffer cache blocks associated with the MTE are written to disk before the MTE is reallocated.

Paging within XMF occurs in two ways: reading is performed on demand, while writing is done asynchronously by a separate XMF daemon process. XMF uses an LRU-like clock algorithm [Corbato69, Babaoğlu81] to manage file buffer cache block replacement, supports delayed writing and block read-ahead, and clusters pages together during I/O to minimize various overheads. Together, these techniques emulate the strategies used in HP-UX.

XMF was embedded in release 1.1 of the HP-UX system. It adds 16 C-language source files, 1 small assembler file, and 6 header files to the HP-UX sources. These files contain approximately 3000 lines of code, and increase the static size of the kernel by about 10%

(124 KB).

3.2 I/O in XMF and HP-UX

This section describes the differences in the execution of READ calls between HP-UX and XMF. READs and WRITEs are identical except that a WRITE may sometimes cause a file to be extended, in which case a new disk block has to be allocated and zero-filled.

A READ system call takes three arguments: a file descriptor; a pointer to a region of memory where the result should be placed; and the number of bytes to transfer. The file descriptor indexes a process-local table, to retrieve a pointer into the system-wide file table that describes all open files. In each file table entry is an inode pointer and an offset where the next READ/WRITE operation will begin. READs and WRITEs access sequential locations in the file unless the file offset is explicitly changed through SEEK or LSEEK system calls.

The differences between READs on XMF and HP-UX happen during the translation from a file offset to a location in the file buffer cache:

- For each READ operation, HP-UX first translates the file offset into a physical disk block number, using the information in the inode and its indirect blocks. It then uses this to index the file buffer cache via a hash table. Physical block numbers are used because the file buffer cache holds some data (inode indirect blocks) that are not part of any file and so have no other address.

- XMF assigns a portion of the processor's virtual address space to represent the file when it is first opened. The virtual address of the start of the file is stored in the system file table. A file's buffer cache address is simply this value plus the file's byte offset. A simple data-copying loop performs the I/O transfer, and the virtual memory hardware handles protection and does the necessary address translations:
  - If the virtual address is stored in the TLB, the translation occurs during a single machine cycle.
  - If the address is not in the hardware TLB, the procesor takes a *TLB fault* trap. If a search in the PDIR data structure finds a descriptor for the relevant page, the TLB is reloaded and the faulting instruction restarted. This is analagous to an HP-UX cache lookup when all the needed indirect blocks are in the file buffer cache.
  - If the PDIR search fails, it means that the required data are not in physical memory and a *page fault* occurs. This is equivalent to HP-UX finding that the needed block is not in its file buffer cache. The faulting virtual address is hashed to locate the relevant VSO, which points to the MTE, which in turn points to the ASO and hence the inode. The logical file block number is then calculated, and used to locate the physical disk block through the inode and indirect blocks in the usual way.

  XMF keeps only file data blocks in its file buffer cache; the indirect blocks are stored in an area of memory managed as a vestigial HP-UX file buffer cache, addressed by physical disk block. This ensures compatibility with the HP-UX disk format.

The main benefit of the XMF scheme is that finding data that are already in the cache is *much* faster than with a software search algorithm (especially if the necessary translation is already in the TLB). In addition, there is no need for an inode indirect block to be in memory for the data blocks it describes to be accessible.

3.3 Modifications to HP-UX

Preliminary measurements indicated that HP-UX performed less well in certain circumstances than we had anticipated. Further investigation showed a number of places where the cache replacement policies or algorithms used by HP-UX were inferior to those used in XMF.[3] Since our goal was to compare cache search mechanisms, not buffer management schemes, we modified HP-UX slightly to make its policies the same as in XMF and re-ran the measurements. In all cases the modified version of HP-UX did better than the old one, so the performance numbers from the new one are used in what follows. Appendix 1 describes the changes in more detail.

4 The experiments

Once the XMF implementation had been completed, attention turned to testing the validity of our hypotheses. Among the issues we investigated were:

- How well a hardware-assisted file buffer cache scheme performed in comparison with a software one.

- How changes in the file buffer cache size and access time affected the performance of file system operations.

- How file system efficiency affected overall system performance.

- How different buffer management strategies affected the hardware processor caches and TLBs.

- How much overhead was added to OPEN/CLOSE system calls. (XMF builds up and tears down additional tables during these calls.)

[3] Several of the HP-UX limitations we identified have been lifted in more recent releases.

The hardware used for the XMF experiments reported in this paper was the HP 9000 Series 840 processor [Fotland87], the first released product in the HP-PA family. It is constructed from TTL logic, operates at a sustained rate of 4.5 MIPS (peaking at 8 MIPS), implements a 48-bit subset of the full HP-PA 64-bit address space, and has a pair of 2048-entry direct-mapped TLBs: one for data, the other for instructions.

One of the test systems had an HP 93562A hardware coprocessor (the "Analyzer Card"), capable of monitoring various hardware functions non-invasively. It was used with XMF to measure the path length of various file system operations, and to collect hardware cache and TLB access and miss counts.

Three utility programs were developed to exercise the file system operations: one for READ, one for WRITE, and one for OPEN/CREATE and CLOSE. Each executed many consecutive system calls, with parameters determined by a set of user-specified options. Larger-scale application benchmarks were used to determine overall performance differences between XMF and HP-UX.

The testing procedure was designed to be highly repeatable: the benchmark programs were run under identical circumstances without any operator intervention. The system was rebooted between each benchmark run; no background daemons were present during the tests. XMF and HP-UX have different buffer-flushing mechanisms, and the init process periodically flushes data to disk, so we disabled the SYNC system call entry point. XMF and HP-UX tests were run alternately using the same file systems on the same disk pack to minimize the effects of changes in disk layouts between runs.

Most experiments were repeated several times. The mean performance numbers are reported here. Averaged over all the experiments, the data variability (the standard deviation divided by the mean value) had a mean value of 2.4% and a maximum of 11%. Appendix II provides more information on the statistical properties of the measurements.

5 Micro performance results

This section summarizes the results obtained from measuring "micro-level" performance parameters. It presents instruction counts for READs from blocks in the buffer cache and on disk; READ bandwidth; and the relative costs of OPEN, CLOSE, and CREATE system calls. Section 6 presents the results from running "macro-level" application-level benchmarks.

5.1 Reading from the file buffer cache

READ results were essentially identical to WRITE results, so only the latter are presented here. [Braunstein89] includes both sets of data.

Small READ/WRITE transfers to and from the buffer cache are dominated by the system call overhead and the cost of locating data in the buffer cache, while copying costs dominate larger transfers.

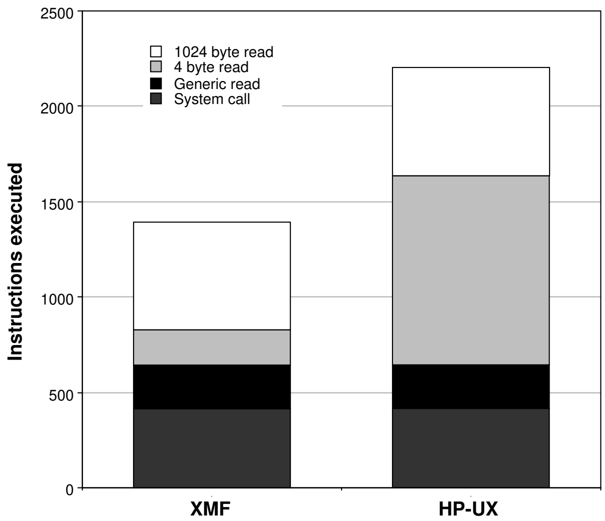

Figure 5: Instructions executed by 4 byte and 1024 byte READ system calls.

Breakdowns of instruction executions during very short (4 byte) and moderate-length (1 KB) READ calls from cache-resident data are shown in Figure 5. (1 KB is the default block size used by the HP-UX stdio library.) The instruction count labelled "4 byte read" is mostly the cost of searching the file buffer cache since the data trsnsfer is so small. This is a substantial part of the total for small READ sizes (22% for XMF, 61% for HP-UX). All the other costs are identical between the two systems: the system call (itself 50% of the total for XMF, 25% for HP-UX); the "generic read" code; and the difference between the 4 byte and 1024 byte transfer sizes, which is the cost of copying the extra data into the user process.

Figure 6 shows how the request size affects the total cost of a READ: byte copying only becomes a dominant component above about 2 KB transfer sizes.[4] Figure 6 might suggest that XMF only performs significantly better at small transfer sizes. However, a plot of the ratio of the XMF and HP-UX instruction counts (Figure 7) shows that XMF offers significant advantage at all transfer sizes. The XMF advantage is smallest at the file buffer cache block size (8 KB for this system) because HP-UX does the smallest number of cache block lookups per transfer here. Even so, XMF is still 20% faster. At larger transfer sizes HP-UX has to perform a lookup for each block because individual blocks are scattered through the file buffer cache, whereas XMF file buffer cache blocks are located in sequential virtual memory locations so that finding the next block's vir-

[4] Note that this graph and several others have logarithmic axes.

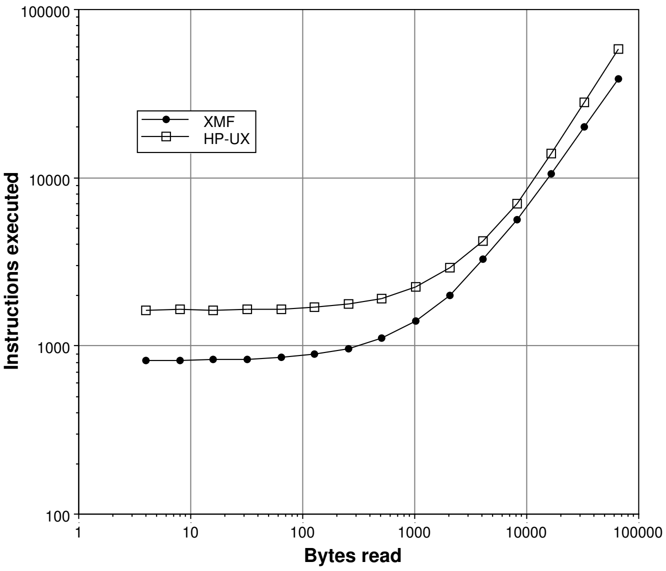

Figure 6: Instructions counts for reading cache-resident data.

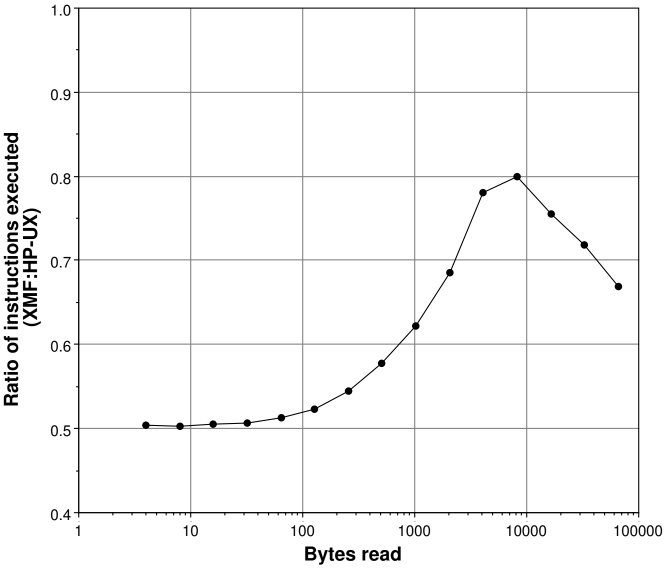

Figure 7: Relative costs of XMF and HP-UX READ system calls; cache-resident data.

tual address is easy.

Figure 8 shows the effect on read bandwidth of varying the file buffer cache size. The test employed 8 KB READs of a 1.5 MB file known to be in the file buffer cache, and the results demonstrate both higher and more consistent bandwidth on XMF as the file buffer cache size is changed. As expected, the bandwidth does not depend significantly on the buffer cache size for XMF (the standard deviations are 0.07 for XMF, 0.09 for HP-UX). The high variance in the HP-UX times is

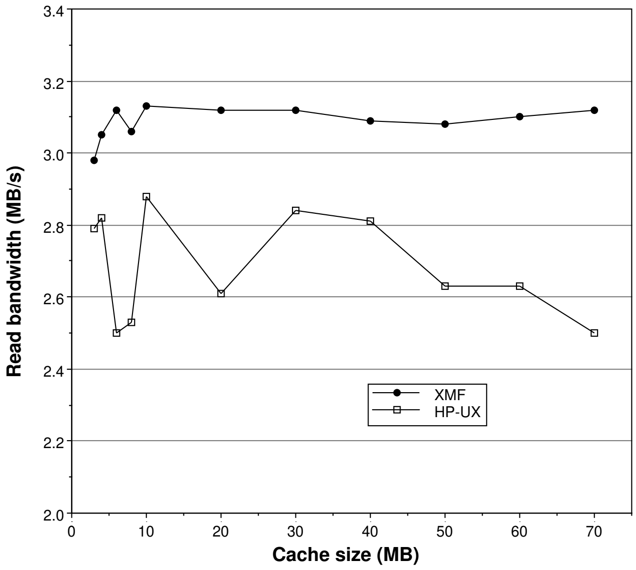

Figure 8: Read bandwidth against file buffer cache size.

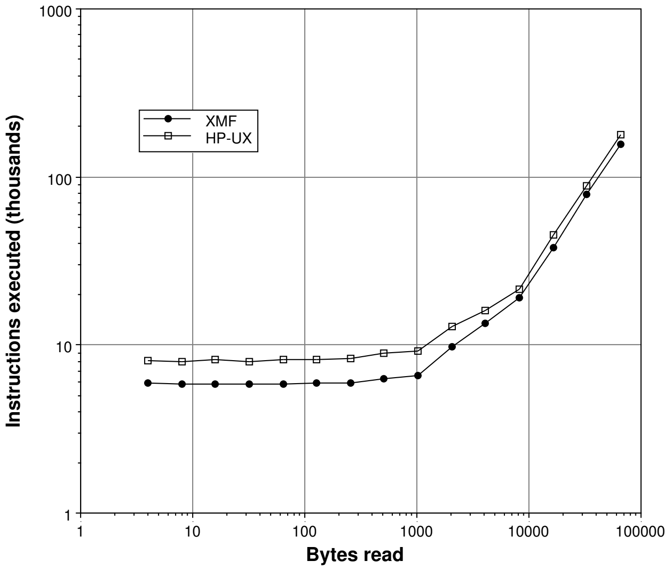

Figure 9: Instruction counts for reading disk-resident data.

discussed further in section 6.2.

5.2 Reading from disk

When the information being requested is not in the file buffer cache, the majority of the time to service a READ request is spent retrieving the data from disk. The READ operation searches the file buffer cache for the block, doesn't find it, allocates a new buffer, brings the block in from the disk, finds the new block (again), and then copies the data to the user process. Figure 9 shows the effect of this on the total instruction count for READ calls. As expected, the I/O subsystem costs

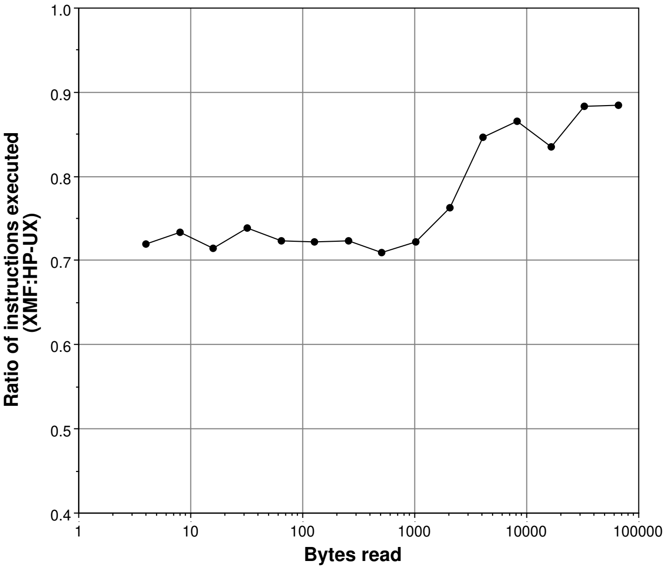

Figure 10: Relative costs of XMF and HP-UX READ system calls; disk-resident data.

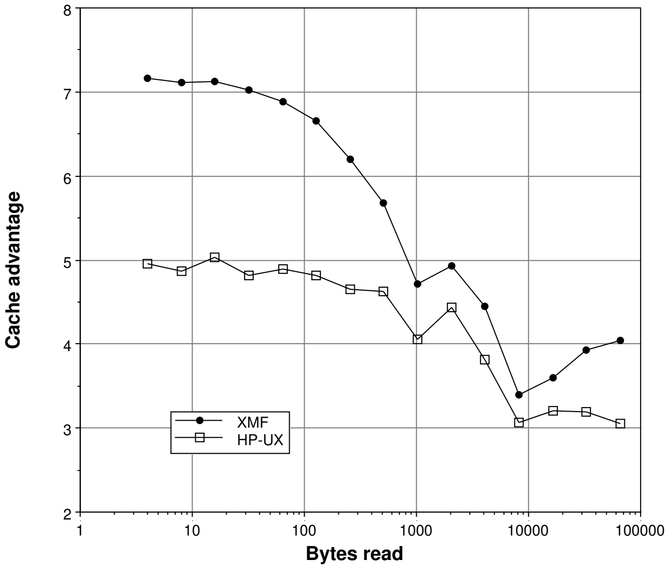

Figure 11: Cache advantages of XMF and HP-UX during READ system calls.

dominate for small transfers: the absolute number of instructions executed is about five times greater than when the disk driver is not called. Despite this, copying costs still dominate for transfers larger than about 4 KB. (The test was constructed in such a way that the read-ahead algorithm was only activated at 8 KB and larger transfer sizes.)

The ratio of the XMF and HP-UX performances is shown in Figure 10. As before, XMF does better for small READ sizes, because both operating systems do a lookup to find that the necessary data block is not in the file buffer cache. A second lookup is also performed after the block is brought in. XMF performs both lookups faster than HP-UX, although the benefit is proportionally less because of the additional costs of the I/O subsystem.

Figure 11 summarizes several of the preceding figures by displaying the relative benefits offered by the caches in XMF and HP-UX. The *cache advantage* is defined as the cost of a cache miss (data retrieved from disk) divided by the cost of a cache hit (data already in the file buffer cache). The larger the cache advantage, the more beneficial a cache is at a given hit rate.

5.3 OPEN/CLOSE performance

In order to support the new READ and WRITE operations, XMF's OPEN, CREATE [5] and CLOSE calls have to manage additional tables. To assess the cost of this, we measured the instruction path lengths for OPENs, CREATEs, and CLOSEs. There were three scenarios:

1. OPEN: a file is opened multiple times, so that the file's inode is already in the inode table and does not need to be paged from the disk.
2. "CREATE existing": the same file is re-created (opened and truncated to zero length) multiple times, reusing the existing inode and other XMF tables.
3. "CREATE new": a new file name is chosen to force a file to be created from scratch with a newly-allocated inode.

The results are displayed in Figure 12. "CREATE new" shows the effect of the XMF table management costs most clearly because XMF always has to build new entries in its three internal tables. "CREATE existing" is similar to OPEN; both calls can often take advantage of XMF's caching of recently-used internal table entries after a file is closed, which reduces the extra cost to only 5%. In all three cases additional work is required to set up the virtual memory hardware information such as protection attributes.

6 Application performance

Micro benchmarks do not tell the whole story: an important concern is the overall effect XMF has on "real" system use. This section describes the results of testing XMF against a diverse group of HP-UX application programs assembled to provide a workload approximating real-world conditions. Each application was a standard part of the local development environment; the only selection criterion was that the program make some use of

[5] The HP-UX CREATE system call is just a special case of OPEN that truncates the file if it already exists. The system call is actually named creat, of course.

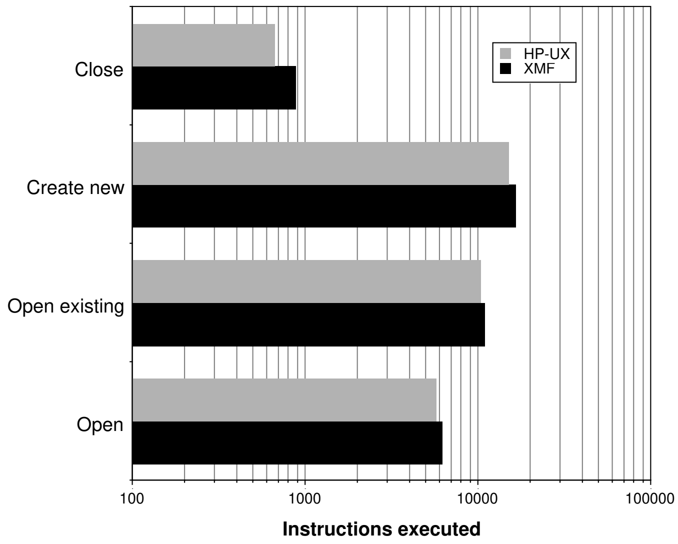

Figure 12: Instructions executed while opening and closing files.

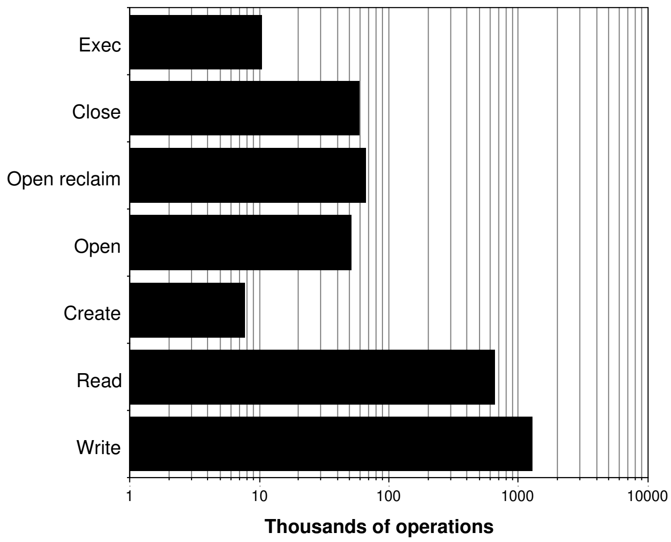

Figure 13: Distribution of file operations in the benchmark suite.

the file system. The applications were exercised using data or scripts also extracted from the local development environment. We believe that these examples are representative of more general classes of applications.

This section discusses the effects on these benchmarks of varying the file buffer cache size, XMF's impact on the TLB, and background paging activity.

The following applications were used to put together our "representative" workload. Each is followed by its approximate running time on an HP-UX system with a 10 MB file buffer cache:

1. HP-UX dependencies: build the source code dependencies for the HP-UX kernel makefile (6:48 minutes; OPEN/CLOSE-intensive).

2. diff: compare two identical 1.5 MB files (14 seconds; CPU-intensive).

3. latex: format a 9 page document from a 25 KB source file (42 seconds; many small READs).

4. sort: sort a 1.5 MB file using an I/O-intensive tape-sort algorithm (12 seconds).

5. Compile HP-UX: generate an optimized HP-UX kernel (1:14 hours; CPU-intensive, OPEN/CLOSE-intensive).

6. Large-system build: compile a very large Pascal-based software system with 18.5 MB of source files and 50 MB of included files (20.6 hours; two threads running in parallel, CPU-intensive).

A *summary benchmark* was assembled using the first four of these, the micro benchmarks described above, and some data analysis scripts used to examine the latter's output. The summary benchmark contained many executions of each application or micro benchmark; depending on the operating system and buffer cache size, it took between 98 and 123 minutes to run. Figure 13 shows the dynamic distribution of file system operations emitted during one of these benchmark runs.

6.1 Effects of file buffer cache size

The machine available for testing had 96 MB of physical memory. By configuring different HP-UX and XMF kernels, the effects of file buffer cache sizes of between 3 MB and 70 MB were investigated. In all cases, sufficient user process memory was available that paging and swapping activities were minimal, and constant across configurations.

Figure 14 shows the running time of the summary benchmark at several different file buffer cache sizes. In all cases XMF did better than HP-UX. User CPU time was 17.6 minutes on both systems, with a variability of 0.3%. System time was 34.3 minutes (variability 1.2%) for XMF, 42.6 minutes (1.3%) for HP-UX. Although the XMF system time does not include the cost of the background XMF virtual memory clock daemon, this cost was close to zero at cache sizes beyond 50 MB because the daemon never needed to run (section 6.3).

Increasing the file buffer cache size beyond about 8 MB for XMF (20 MB for HP-UX) had little effect on the elapsed time. The two curves of Figure 14 do not have knees at identical points, which suggests that we were not entirely succcessful in making the two cache replacement policies identical. Although this does not invalidate our primary hypotheses (XMF shows an 8.3% performance improvement at large cache sizes, beyond the

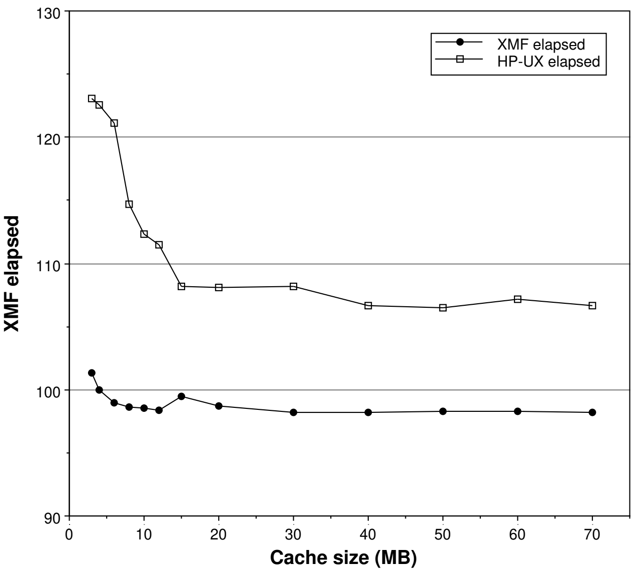

Figure 14: Summary benchmark execution times by file buffer cache size.

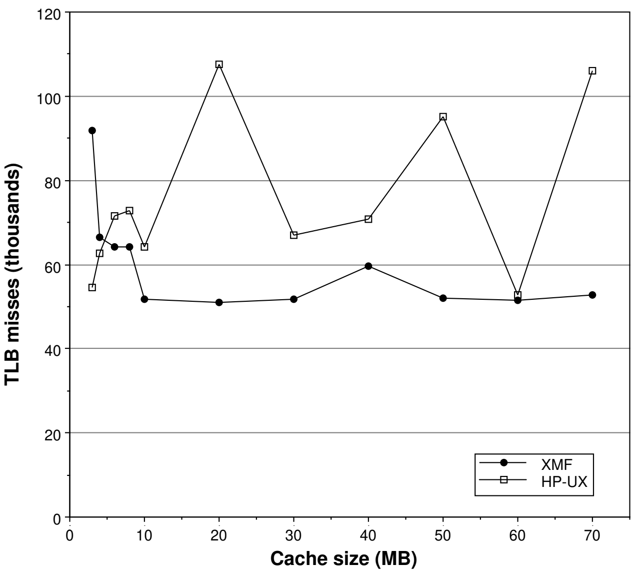

Figure 15: TLB misses: sort benchmark.

knees in both curves), it did cause us to investigate the behavior further. It turns out that HP-UX does less well at smaller cache sizes because it writes unnecessarily many dirty blocks to disk when searching for a block to evict from the buffer cache. XMF does not exhibit this effect because its clock algorithm maintains a pool of free blocks by reclaiming just the blocks it needs. More information on this is presented in Appendix I.

We were not able to construct a meaningful benchmark that significantly stressed very large ($\geq$ 30MB) buffer caches, so our original hypothesis that the XMF technique would scale better to this regime remains unsubstantiated. (Incidentally, this supports the observations of [Ousterhout85].) Even the large Pascal compilation benchmark shows no cache-size related effects, despite its 69 MB of files—perhaps because the Pascal compiler with optimization turned on is computationally intensive (user mode CPU time accounted for 83% of the total time). For all cache sizes between 6 and 40 MB, the benchmark took 20.6 hours to execute on HP-UX, 19.1 hours on XMF—a savings of about 7%.

6.2 Effects on the translation lookaside buffer

We initially hypothesized that XMF would decrease the effectiveness of the hardware virtual memory assist structures (e.g. the TLB) because it would be actively using a larger virtual address space. To investigate whether this was correct, the Analyzer Card was used to measure data TLB accesses and misses during several benchmark runs. In the first test, the HP-UX sort utility was used to sort a 1.5 MB file. HP-UX averaged some 14% more TLB accesses than XMF to perform the same amount of useful work, so one would expect

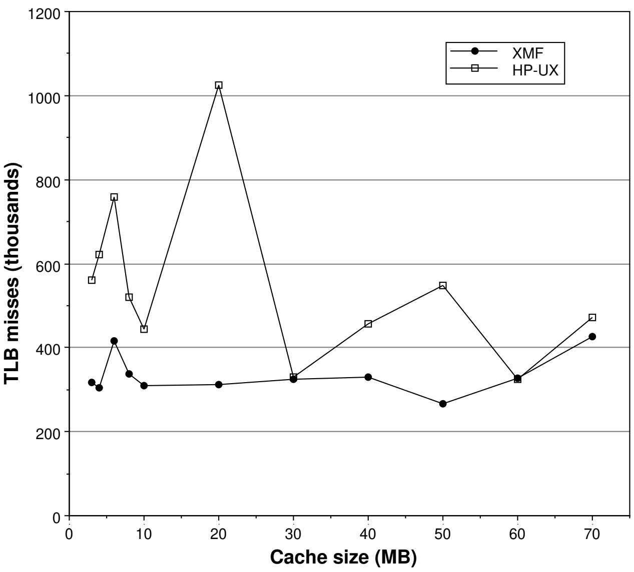

Figure 16: TLB misses: HP-UX dependencies benchmark.

HP-UX to have about 14% more TLB misses than XMF. However, as shown in Figure 15, the number of HP-UX TLB misses varied greatly while the XMF miss counts were fairly constant.

A similar pattern was seen while building the HP-UX dependency list (Figure 16). In this case the number of TLB accesses differs between HP-UX and XMF by less than 1%, but as before, the numbers of TLB misses differ by more than this.

Three other experiments repeatably showed similar results: the mean variability was less than 2% for XMF and less than 6% for HP-UX. These differences in miss rates can have significant effects: for the READ-bandwidth benchmark (Figure 8), the mean difference between the number of TLB misses for XMF and HP-UX accounts for roughly 32 ms of execution time—about 5% of the elapsed execution time, or 0.14 MB/s of bandwidth.

We conjecture that the greater TLB miss counts shown by HP-UX and their higher variances result from three main factors:

1. XMF instruction counts were consistently lower than those for HP-UX, so a similar TLB miss rate would produce fewer TLB misses.

2. The hardware TLB hashing algorithms use some of the high-order address bits, which varied a lot between files for XMF but not much for HP-UX, thereby increasing the relative likelihood of collisions for the latter—effectively, XMF was "spreading out" its references over a larger hash input domain than HP-UX.

3. The contiguous virtual address accesses by XMF for a single file minimized collisions on consecutive READs and WRITEs because the hardware TLB hash algorithm incorporates the low-order bits of the page number.

An experiment to confirm or repudiate these conjectures could be constructed by analysing a trace of all the TLB misses during a benchmark run, and then looking for collisions caused by consecutive accesses, "hot spots" in application or kernel data structures, or the buffer cache itself.

6.3 Background process overheads

The overhead incurred by the XMF background virtual memory daemon was measured by running a processor-intensive benchmark that did no file system I/O (dhrystone). Tests were performed with and without the background virtual memory daemon enabled, although in both cases XMF ran various load calculations 100 times per second. The overhead was quite small: less than 0.3% (the variability of the data).

The amount of physical disk I/O (paging) done by XMF during an early version of the summary benchmark runs is shown in Figure 17. The pageout count reaches a fixed minimum because the benchmark suite explicitly flushes several files to disk. The miss rates for the file buffer cache (the number of disk IOs divided by the number of READ and WRITE system calls) were as follows (the high values for HP-UX at 6 MB are discussed further in Appendix I):

	XMF		HP-UX	
	6 MB	70 MB	6 MB	70 MB
READs	8.8%	8.6%	9.3%	8.6%
WRITEs	2.1%	2.0%	5.2%	2.1%
both	4.4%	4.3%	6.6%	4.3%

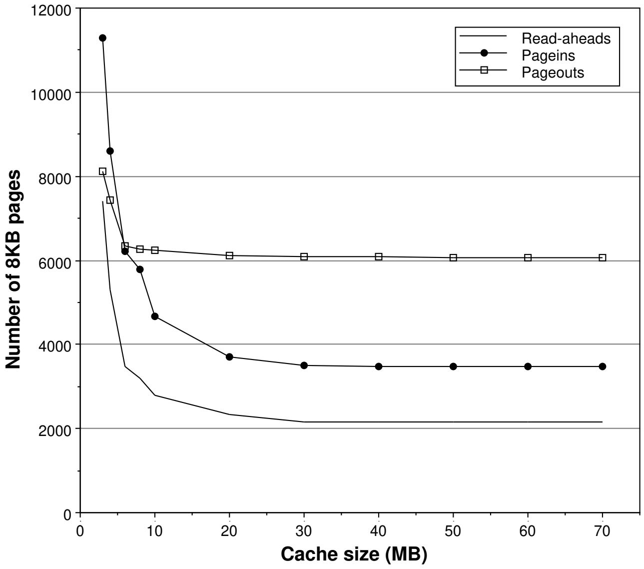

Figure 17: Paging statistics for XMF.

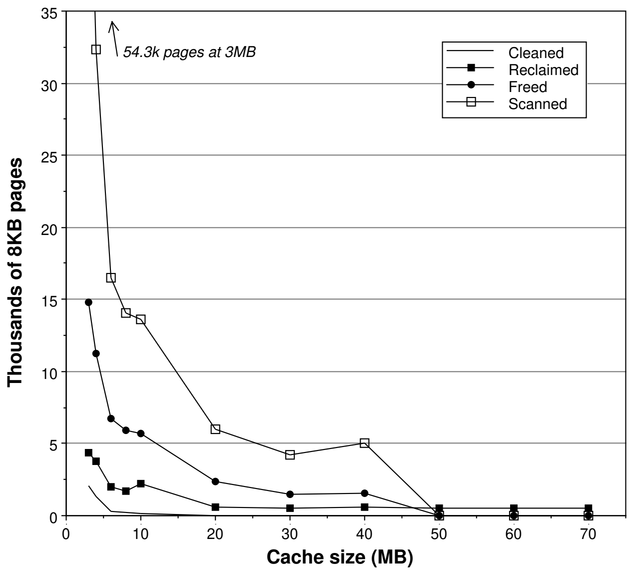

Figure 18: XMF virtual memory daemon activity during a summary benchmark run.

Other data indicate that the read-ahead algorithm only retrieved 5 more blocks from disk than were used for data transfers, confirming that the simple algorithm used performs very well.

Figure 18 shows the activities performed by the background XMF clock daemon over a range of file buffer cache sizes. We did not devise a convincing mechanism to determine how much time the daemon spent doing this work.

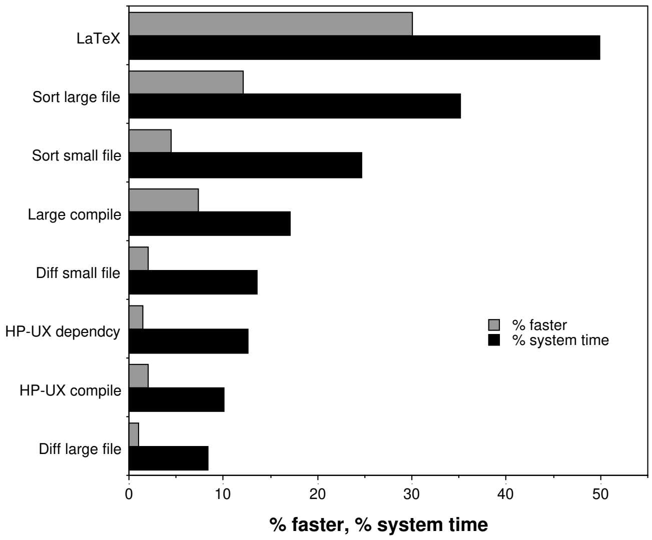

Figure 19: Percentage of the elapsed time spent in system mode against percent speedup when running on XMF.

7 Analysis

Improving various HP-UX algorithms lead to a 10–17% improvement in overall HP-UX performance. Even with all these changes, XMF outperformed the resulting HP-UX system. The following summary of the data reflects the results of comparing XMF to the improved HP-UX:

- Small READs and WRITEs ($\leq$1 KB) under XMF execute about 50% fewer instructions than HP-UX when the data are in the file buffer cache, and about 25% fewer when the data are on the disk.
- Large READs and WRITEs ($\geq$8 KB) under XMF execute about 20% fewer instructions than HP-UX when the data are in the file buffer cache, and about 10% fewer when the data are on the disk.
- For many applications, the time spent in system mode under XMF is about 50% of that of HP-UX. Since the system mode time for our benchmarks represented about 15–30% of the total time, the overall improvement was about 7–15%.
- OPEN/CREATE and CLOSE calls are slightly slower under XMF than HP-UX. This seems to have little effect on overall system performance unless the benchmark accesses a lot of very small files.
- XMF performs much better than HP-UX for file buffer cache sizes less than 10 MB. This is true for all of the HP-UX versions, but it is especially true for the unmodified HP-UX.
- XMF causes fewer data TLB faults than HP-UX.

The likely benefits from running an application on XMF rather than HP-UX can be accounted for by a reasonably simple model that characterizes the application's behavior along the following axes:

- The amount of file-system activity. For the file-system-intensive applications used as benchmarks in this work, the XMF benefit is roughly proportional to the amount of time they spend in system mode (figure 19), although this will not be true of arbitrary applications.
- The I/O transfer sizes. The smaller the transfers, the greater the XMF advantage.
- The ratio of READ and WRITE system calls to OPEN and CLOSE calls. The latter are more expensive in XMF:
  - the summary benchmark has a 33:1 ratio—XMF showed an improvement of 8.3% at large buffer cache sizes (98.5 minutes versus 107.4 minutes);
  - the HP-UX dependencies benchmark has a 4.4:1 ratio—XMF displayed no overall advantage (the benchmark takes a mean of 400 seconds on both systems).

Other parameters are essentially independent of the system used. For example, the relative performance of READs and WRITEs did not change significantly in XMF by comparison to HP-UX. Finally, performance is influenced by the amount of data used by the application. For example, all the data accessed by the latex benchmark fitted into the buffer cache, so its running time was independent of the buffer cache size.

8 Conclusions

This paper has presented a characterization of two methods of accessing data in a file buffer cache under a variety of memory configurations and application mixes.

The methodology used was to propose a set of hypotheses and then conduct controlled experiments to test them, using two otherwise-identical systems, so that conclusions could be drawn directly from the data. Despite careful preparation and execution, we were repeatedly surprised by subtle but important effects we had not controlled for. A number of iterations of measurement and implementation were required to eliminate or understand these effects.

Almost all of the initial hypotheses proved correct, with two exceptions: the better scaling of XMF to very large buffer caches is still undecided, and TLB-related overheads were less on XMF than on HP-UX. Virtual memory hardware assists for managing file caches *do* offer significant performance advantages in almost every case we measured.

We believe that our results are more widely applicable than the deliberately narrow focus of this study might at first suggest.

For example, we feel that this work demonstrates the merits of using application-specific information (e.g. that read-ahead is a beneficial strategy for file data) in conjunction with a general mechanism (virtual memory management).

The XMF design is a particular example of searching system data structures by associative lookup. Hardware to help with this is commonly available in the form of processor virtual memory assists, such as TLBs. In our experience, it was quite easy to take advantage of it.

Compared with HP-UX, XMF is virtually untuned, yet it performs at least as well—sometimes much better. XMF has simple, straightforward mechanisms that allow it to do the common operations well. These also eased changing it in the course of experimentation. The complexity of the HP-UX implementation made such exercises hazardous, and increased the difficulty of understanding its behavior.

In summary, the use of hardware virtual assists for file buffer cache lookup in XMF resulted in a straightforward, simple implementation that performed at least as well as—and often better than—our improved HP-UX, and much better than the original version.

Appendix I: changes to HP-UX

The first portion of this appendix describes the changes we made to HP-UX's management of its buffer cache; they may prove useful for other 4.2BSD-based systems. The second elaborates on the discrepancy in the knees of the performance curves in Figure 14.

Improvements to the buffer cache algorithms

We found three problem areas that prevented HP-UX from performing as well as XMF:

- The division of file buffers between two internal queues (the LRU and AGE queues) caused long-lived blocks (like inodes and indirect blocks) to fill up the LRU queue. Only the AGE queue was used to allocate new data blocks, so the cache behaved as if it was much smaller than it really was. We merged the two queues into a single structure with true LRU behavior, so each could take full advantage of the available buffer pool.

- The size of the hash table used to index the buffer cache was fixed at kernel compilation time at a very small value (63 entries). The collision chains were searched linearly, so performance decreased as the number of buffers in the file buffer cache increased (with a 70 MB cache, the average chain was 220 entries long). We changed the code to allocate and size the hash table only after the number of file buffer cache entries was known, to make the average collision chain length about 0.5.

- When the last byte in a file buffer was written, the file system predicted that the block would not be referenced again and forcibly flushed the buffer to disk. Applications that wrote data a full buffer at a time would pay a performance penalty, and short-lived data (such as temporary files) were likely to be flushed unnecessarily to disk. We removed this code and implemented a true delayed-write strategy.

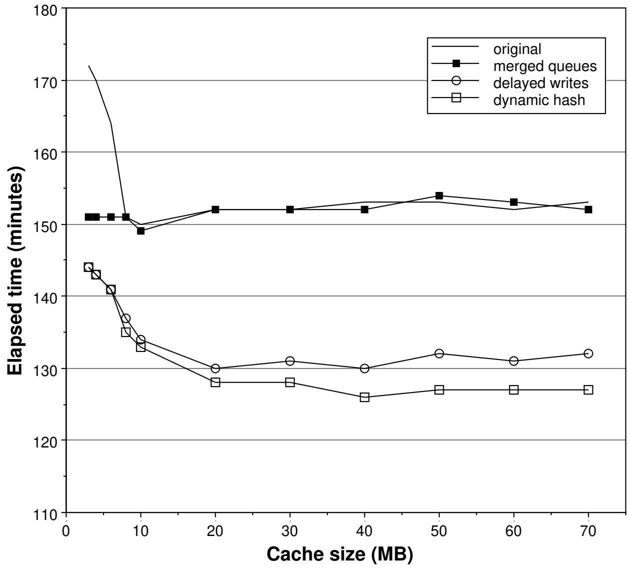

Figure 20: Performance comparisons of different HP-UX versions.

Figure 20 shows the performance of HP-UX variants incorporating the changes described above (the test was conducted with an early variant of the summary benchmark):

1. Unaltered HP-UX ("original").

2. Merged LRU/AGE queues ("merged queues").

3. Merged LRU/AGE queues and true delayed writes ("delayed writes").

4. Merged LRU/AGE queues, true delayed writes, and the hash table size computed dynamically at boot time ("dynamic hash").

In original HP-UX, the static hash table size caused an increase in execution time for large file buffer caches (about 4% between 3 MB and 70 MB). Merging the LRU/AGE queues allowed file buffer caches between 3 MB and 10 MB to perform better under the tested workload. Changing the file buffer cache to use a true delayed-write algorithm increased overall performance by up to

14%. Overall, the three algorithm changes resulted in improvements of 17% using a 3 MB file buffer cache, 10% using a 10 MB file buffer cache, and 17% using file buffer caches 20 MB and greater.

Less than a dozen lines of code were changed to achieve these benefits.

Cache replacement policies

The difference in the cache sizes for the knees of the two curves in Figure 14 indicates that XMF is not simply doing cache lookups faster than HP-UX—it's doing something else better as well. We first conjectured that this could be explained simply by different buffer residency times. If some blocks in HP-UX had longer residencies than in XMF, the result would be a smaller effective cache size for HP-UX. We identified two possible mechanisms by which this could happen:

- inode and indirect blocks in HP-UX are treated as part of the single buffer cache (and have a high likelihood of remaining in memory, so reducing the effective buffer cache size), while in XMF they are confined to a separate portion of the cache and (incidentally) accessed less frequently because of the hardware assists;

- the XMF clock daemon only approximates HP-UX's true LRU algorithm (it does best at very small and very large memory sizes).

A test for this hypothesis would be to log the cache entry and exit times of each block, and then compare the residency profiles for the two systems. Such tests would involve many modifications to the kernel sources, which was something we were loath to do when this behavior came to light. Instead, we ran an experiment that logged all physical disk traffic during runs of the summary benchmark over a range of cache sizes.

The results are shown in Figure 21 (the effects of the logging, swapping and paging traffic have been subtracted out). If the replacement policies were the same, the curves for XMF and HP-UX would be identical. Instead, as the buffer cache size gets smaller, the number of disk writes issued by HP-UX climbs much faster than for XMF, even though the number of disk reads rises only slightly. For each extra disk read HP-UX does compared with XMF, 12 times as many extra writes occur. The resulting knee in the total I/O curve is at a similar point to the one for Figure 14.

Looking at the distribution of inter-arrival times of the requests at the disk driver showed that the additional writes were coming in *clumps* of requests, each request arriving less than 10 ms after its predecessor (Figure 22.) The peak in the back right hand corner of the graph represents this anomalous HP-UX write traffic. Such clumps often contained *runs* of physically contiguous disk blocks: at a cache size of 6 MB, the mean run

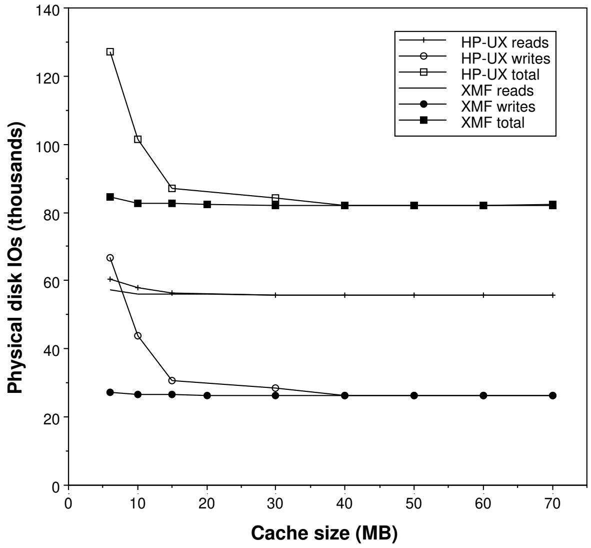

Figure 21: Physical disk traffic during the summary benchmark.

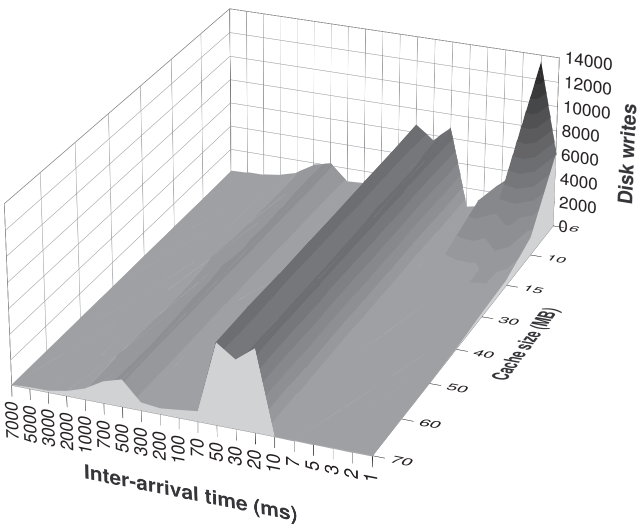

Figure 22: HP-UX disk writes as a function of buffer cache size and inter-arrival time at the device driver. The vertical axis is the number of disk writes during the benchmark; the foreground one is a set of time buckets.

size was 15 blocks, with 91% of the clumped requests being found in runs of 5 or more blocks. The largest such run we observed was of 244 contiguous blocks, or 1.9 MB of data.

We tracked the effect down to an algorithm in the HP-UX function getnewbuf, which is called to select a block to toss out of the cache. It scans down the LRU-ordered list of eligible blocks looking for a clean block to be its victim for replacement. If it encounters any dirty blocks as it goes down the list, it schedules them all for writing (to clean them). It continues this until it finds a clean block, or until there are no more blocks in the list, at which point it waits until one of the writes completes. Since the list sometimes contained long lists of dirty blocks, often associated with serial WRITEs to the same file, the effect was to generate large number of writes with very short inter-arrival times.

It might be thought that these writes would be necessary anyway, and pushing the dirty blocks out to disk would not change the total number of I/Os. However, if the cache is big enough, short-lived temporary files may never be written to disk: their blocks get marked empty when the file is deleted. Only at smaller cache sizes will some such blocks need to be reclaimed and their contents pushed to disk. The result is that HP-UX sometimes needlessly cleans many more blocks than are immediately required when searching for a single buffer to replace.

It is not immediately clear what a better policy would be for it: skipping over dirty blocks would sacrifice true LRU behavior; using the first block, and waiting for a synchronous write to clean it if was dirty, would serialize the cleaning and get-new-block operations. This behavior is a fundamental difference between the two page-cleaning algorithms: XMF avoids the issue by using the clock algorithm to maintain a free pool of clean blocks, cleaning a few blocks each time it runs.

In summary, because the (postulated) longer residency times of inode and indirect blocks effectively reduces the HP-UX cache size, it needs to reclaim a few more blocks than XMF at a given physical cache size. (The slight increase in physical reads for HP-UX by comparison with XMF shown in Figure 21 supports this hypothesis.) When this happens, the getnewbuf algorithm can result in large increases in disk write counts, and HP-UX's performance suffers accordingly.

Appendix II: statistical details

This appendix presents some statistical information on the data presented in this paper.

The table below shows the number of times each experiment was run and the mean and maximum variances that were measured for it. It also shows the number of "outlier" data points that were discarded as a percentage of the total points for the whole graph (e.g. the occasional instruction count measurement that coincided with an external interrupt and was very much larger than the remainder of the data). Figures 13, 17, 18, 19, 20, 21, and 22 show data that was gathered from a single experimental run for each data point.

			XMF		HP-UX	
Figure number	Runs	% Discards	% Variances Mean	% Variances Max	% Variances Mean	% Variances Max
5	30	6	—	0.6	—	0.6
6	15	3	1.3	4.3	2.7	7.9
8	10	10	2.1	2.8	2.2	3.3
9	5	6	3.2	5.9	3.1	10.7
12	15	3	1.7	3.9	3.0	3.9
14	2	0	0.2	0.4	1.0	1.4
15	6	0	1.5	4.6	5.5	9.9
16	6	0	1.8	2.6	4.6	11.1

References

[Babaoğlu81] Ö. Babaoğlu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. *Proceedings 8th Symposium on Operating System Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **15**(5):78–86, December 1981.

[Bensoussan72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The MULTICS virtual memory, concepts and design. *Communications of the ACM*, **15**(5):308–18, May 1972.

[Braunstein89] A. S. Braunstein. *File buffer cache design in computers with large physical memories.* Masters thesis. Massachusetts Institute of Technology, February 1989. Published as Hewlett-Packard Laboratories Technical Report HPL–DSD–89–6.

[Busch87] J. R. Busch, A. J. Kondoff, and D. Ouye. MPE XL: the operating system for HP's next generation of commercial computer systems. *Hewlett-Packard Journal*, **38**(11):68–86, December 1987.

[Chang88] A. Chang and M. F. Mergen. 801 storage: architecture and programming. *ACM Transactions on Computer Systems*, **6**(1):28–50, February 1988.

[Cheriton87] D. Cheriton. *Effective use of large RAM diskless workstations with the V virtual memory system.* Technical report. Department of Computer Science, Stanford University, 1987.

[Clegg86] F. W. Clegg, G. S.-F. Ho, S. R. Kusmer, and J. R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal*, **37**(12):4–22, December 1986.

[Corbato69] F. J. Corbato. A paging experiment with the Multics system. In *In Honor of P. M. Morse*, pages 217–28. MIT Press, 1969.

[Feder84] J. Feder. The evolution of UNIX system performance. *AT&T Bell Laboratories Technical Journal*, **63**(8 part 2):1791–814, October 1984.

[Fitzgerald86] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, **4**(2):147–77. *Presented at 10th Symposium on Operating System Principles* (Orcas Island, Washington, December 1985.), May 1986.

[Fotland87] D. A. Fotland, J. F. Shelton, W. R. Bryg, R. V. La Fetra, S. I. Boschma, A. S. Yeh, and E. M. Jacobs. Hardware design of the first HP Precision Architecture computers. *Hewlett-Packard Journal*, **38**(3):4–17, March 1987.

[Gifford88] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, **31**(3):288–98, March 1988.

[Henry78] G. G. Henry. Introduction to IBM System/38 architecture. In *IBM System/38 Technical Developments*. IBM Corporation, Atlanta, GA, 1978.

[Holt87] D. Holt, December 1987. Private communication.

[Kilburn62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage. *IRE Transactions on Electronic Computers*, **EC–11**(2):223–35, April 1962.

[Leach83] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communication*, **SAC-1**(5):842–57, November 1983.

[Mahon86] M. J. Mahon, R. B.-L. Lee, T. C. Miller, J. C. Huck, and W. R. Bryg. Hewlett-Packard Precision Architecture: the processor. *Hewlett-Packard Journal*, **37**(8):4–21, August 1986.

[Nelson88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, **6**(1):134–54, February 1988.

[OQuin86] J. C. O'Quin, J. T. O'Quin, M. D. Rogers, and T. A. Smith. Design of the IBM RT PC Virtual Memory manager. In F. Waters, editor, *IBM RT Personal Computer Technology*, pages 126–30. IBM Engineering Systems Products, Milford, Connecticut, Form No SA23–1057, 1986.

[Organick72] E. I. Organick. *The MULTICS System: an Examination of its Structure*. MIT Press, Cambridge, Mass, 1972.

[Ousterhout85] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings 10th Symposium on Operating System Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[Ousterhout88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, **21**(2):23–36, February 1988.

[Redell80] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: an operating system for a personal computer. *Communications of the ACM*, **23**(2):81–92. Presented at the *7th Symposium on Operating System Principles* (December 1979), February 1980.

[Satyanarayanan85] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: principles and design. *Proceedings 10th Symposium on Operating System Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):35–50, December 1985.

[Schroeder85] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. *Proceedings 10th Symposium on Operating System Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):25–34, December 1985.

[Simpson87] R. O. Simpson and P. D. Hester. The IBM RT PC ROMP processor and memory management unit architecture. *IBM Systems Journal*, **26**(4):346–60, 1987.

[Swinehart79] D. Swinehart, G. McDaniel, and D. Boggs. WFS: a simple shared file system for a distributed environment. *Proceedings 7th Symposium on Operating System Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **13**(5):9–17, December 1979.

[Tevanian87] A. Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, published as Technical report CMU–CS–88–106. Carnegie-Mellon University, Pittsburgh, Pa, December 1987.

[Tevanian88] A. Tevanian, R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi. *A UNIX interface for shared memory and* *memory mapped files under Mach*. Technical report. Department of Computer Science, Carnegie-Mellon University, 1988.