Application-Controlled Physical Memory using External Page-Cache Management

Kieran Harty and David R. Cheriton
Computer Science Department
Stanford University, CA 94305

Abstract

Next generation computer systems will have gigabytes of physical memory and processors in the 200 MIPS range or higher. While this trend suggests that memory management for most programs will be less of a concern, memory-bound applications such as scientific simulations and database management systems will require more sophisticated memory management support, especially in a multiprogramming environment. Furthermore, new architectures are introducing new complexities between the processor and memory, requiring techniques such as page coloring, variable page sizes and physical placement control.

We describe the design, implementation and evaluation of a virtual memory system that provides application control of physical memory using *external page-cache management*. In this approach, a sophisticated application is able to monitor and control the amount of physical memory it has available for execution, the exact contents of this memory, and the scheduling and nature of page-in and page-out using the abstraction of a page frame cache provided by the kernel. It is also able to handle multiple page sizes and control the specific physical pages it uses. We claim that this approach can significantly improve performance for many memory-bound applications while reducing kernel complexity, yet does not complicate other applications or reduce their performance.

1 Introduction

Next generation computer systems will measure their physical memory in gigabytes, just as current systems are rated in megabytes and previous generation systems were rated in kilobytes. This trend has prompted some to foretell the demise of operating system virtual memory systems and even secondary storage. Yet, secondary storage and networking growth places the effective external data capacities in the terabyte range, maintaining the rough ratio of main to secondary storage that has held for decades. Thus, the *real effect* of the arrival of gigabyte memories is to clearly delineate applications with modest memory requirements from those whose requirements are almost unbounded, such as large-scale simulation, or whose requirements grow proportional to external data capacities, such as data base systems. The increasing speed of processors and the lack of comparable improvement in I/O performance makes the memory system performance a key limiting factor for these demanding applications. With a page fault to secondary storage now costing close to a million instruction times, the "instruction budget" exists to take a more intelligent approach to page management in virtual memory systems.

There are three major problems with current virtual memory systems. Firstly, an application cannot know the amount of physical memory it has available, it is not informed when significant changes are made in the amount of available memory, and it cannot control the specific physical pages it is allocated. Secondly, a program cannot efficiently control the contents of the physical memory allocated to it. Finally, a program cannot easily control the read-ahead, writeback and discarding of pages within its physical memory. Addressing these problems has significant performance benefits for applications, as argued below.

With knowledge of the amount of available physical memory, an application may be able to make an intelligent space-time tradeoff between different algorithms or modes of execution that achieve its desired computation. For example, MP3D [7], a large scale parallel particle simulation based on the Monte-Carlo method, generates a final result based on the averaging of a number of simulation runs. The simulation can be run for a shorter amount of time if it uses many runs with a large number of particles. This application could automatically adjust the number of particles it uses for a run, and thus the amount of memory it requires, based on availability of physical memory. Similarly, a parallel database query processing program [17] can adapt the degree of parallelism it uses, and thus its memory usage, based on memory availability. Finally, a run-time memory management library using garbage collection can adapt the frequency of collections to available physical memory, if this information is available to it.

With control of which specific physical page frames it uses and their virtual memory mapping, an application can optimize for efficient access based on the system memory organization and the application access patterns. For example, in the DASH machine [13], physical memory is distributed, even though the machine provides a consistent shared memory abstraction using a cache consistency protocol. In this type of machine, a large-scale application can allocate page frames to specific portions of the program based on a page frame's physical location in the machine and the expected access to this portion of memory. Similarly, an application can allocate physical pages to virtual pages to minimize mapping collisions in physically addressed caches and TLBs, implementing page coloring [15] on an application-specific basis, taking into account expected data access patterns at run-time.

With control of the portion of its virtual address space mapped to physical memory, an application can operate far more efficiently if it is using a virtual address space that exceeds the size of physical memory. For example, a database management system can ensure that critical pages, such as those containing central indices and directories, are in physical memory. The query optimizer and transaction scheduler can also benefit from knowing which pages are in memory, because the cost of a page fault can significantly increase the overall cost of a query. The latency of a page fault also dramatically extends lock hold time times if locks are held across a fault. With multiprocessor machines, an unfortunate page fault can cost not just the elapsed time of the fault, but that cost multiplied by the number of processes blocked if they also hit the same page, or a lock held the blocked process.

With control of read-ahead, writeback and page discarding, an application can its minimize I/O bandwidth requirements and the effect of I/O latencies on its execution. Scientific computations using large data sets can often predict their data access patterns well in advance, which allows the disk access latency to be overlapped with current computation, if efficient application-directed read-ahead and writeback are supported by the operating system (and the requisite I/O bandwidth is available). For example, the large-scale particle simulation cited above takes approximately 12 seconds to scan its in-memory data of 200 megabytes for each simulated time interval (on a machine with eight 30-MIPS processors [1]). Thus there is ample time to overlap prefetching and writeback if the data does not fit entirely in memory.

Extensions to virtual memory systems, such as page pinning, external pagers [21, 5] and application-program advisory system calls like the Unix [2] *madvise* attempt to address some of these issues, but incompletely and with significant increase in kernel complexity. We are interested in exploring a significantly different modularization of the memory system implementation that both provides application control and reduces kernel complexity.

In this paper, we describe the design, implementation and evaluation of a virtual memory system that provides application control of physical memory using what we call *external page-cache management.* With external page-cache management, the virtual memory system effectively provides the application with one or more physical page caches that the application can manage external to the kernel. In particular, it can know the exact size of the cache in page frames. It can control exactly which page is selected for replacement on a page fault and it can control completely how data is transferred into and out of the page, including selecting read-ahead and writeback. It can also has information about physical addresses, so that it can implement schemes like page coloring and physical placement control.

In essence, the kernel virtual memory system provides a page frame cache for the application to manage, rather than a conventional transparent virtual address space that makes the main memory versus secondary storage division transparent *except for performance.* A default process-level manager provides page-cache management for such applications that do not want to manage their virtual memory,

The next section describes our design as implemented in the V++ kernel. The following section evaluates external page-cache management, drawing on measurements both from the V++ implementation and a simulated database transaction processing system. Section 4 describes related work. We close with a discussion of conclusions and some future directions.

2 External Page-Cache Management in V++

External page-cache management requires new kernel operations and process-level modules to allow process-level management of page frames. We first describe the kernel support, followed by a discussion of application-specific managers. We then discuss the the default manager. Finally we describe the module responsible for global memory allocation. Although this section focuses on the design and implementation of external page-cache management in V++, a new generation of the V distributed system, the basic approach is applicable to other systems, such as Unix.

2.1 Kernel Page Cache Management Support

Kernel page cache management support is provided in V++ as operations on segments. A *segment* is a variable-size address range of zero or more pages, similar to the conventional virtual memory notion of segment [3]. Pages can be added, removed, mapped, unmapped, read and written using segment operations. A parameter to the segment creation call optionally specifies the page size to support machines such as those using the Alpha microprocessor [10] that support multiple page sizes. Segments are used for cached and mapped files, portions of program address spaces (such as the code segment, data segment, etc.) as well as for program address spaces themselves, as illustrated in Figure 1. Referring to Figure 1, a program virtual address space in V++ is a segment that is composed by binding one or more *regions* of other segments. The figure illustrates a virtual address segment with a code, data and stack segments bound into the code, data and stack regions of the address space, respectively. A bound region

[1] Silicon Graphics 4D/380

[2] UNIX is a trademark of AT&T

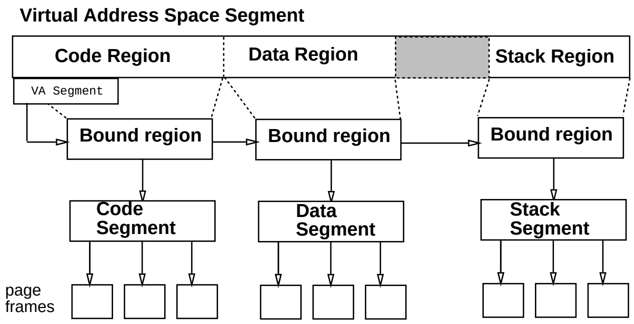

Figure 1: Kernel Implementation of a Virtual Address Space

associates a range of addresses (page-aligned and a multiple of pages) in one segment with an equal-sized range of blocks in another segment so that a memory reference to an address covered by a bound region in first segment is effectively a reference to the corresponding address in the associated bound segment. The binding facilities also support a copy-on-write binding in which pages are effectively bound to a source segment until modified. While this segment structure is similar to other virtual memory designs, the novelty lies in the associated page cache management support.

External page cache management is supported with three significant additions over conventional virtual memory management operations, if one regards the V++ segment as roughly analogous to Unix open files and Mach memory objects. Firstly, an explicit *manager* module is associated with each segment, using the kernel operation,

```
SetSegmentManager( seg, manager )
```

Secondly, the kernel operation

```
MigratePages(srcSeg, dstSeg, srcPage, dstPage,
             pages, sFlgs, cFlgs)
```

moves `pages` page frames from the source segment, starting at `srcPage` to the destination segment, starting at *dstPage*, setting the page flags specified by *sFlgs* and clearing the page flags specified by *cFlgs* for each migrated page frame. A similar kernel operation,

```
ModifyPageFlags(seg, page, pages, sFlgs, cFlgs)
```

modifies the page flags without migrating the page frames. The `MigratePages` and `ModifyPageFlags` operations allow the manager to modify page state flags such as the *dirty* flag in addition to the protection flags accessible with the conventional Unix `mprotect`. Finally, the kernel operation

```
GetPageAttributes( seg, page, pages)
  returns (pageAttributeArray)
```

returns the page flags, and the physical pageframe address, of the specified set of page frames. These operations are used in conjunction with modest extensions of conventional virtual memory facilities, such as the ability to catch page faults at user level, to implement external page cache management.

The segment manager is a module responsible for managing the pages associated with the segment. In particular, when a reference is made to a missing or protected page frame in a segment, the event is communicated to the manager. The manager handles the fault following the sequence illustrated in Figure 2. Referring to this figure, when the

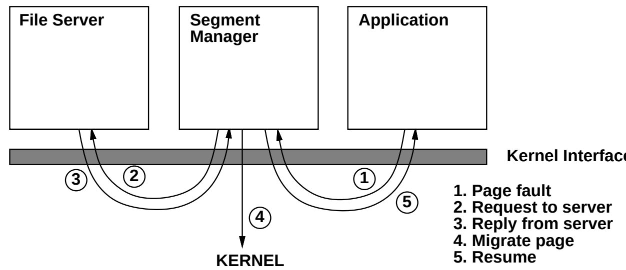

Figure 2: Page Fault Handling with External Page-Cache Management

application references a page frame not present in its address space, it traps to the kernel, which forwards the page fault indication to the manager (step 1). The manager allocates a page frame from another segment (often a *free-page segment*), requests the data for the page frame from the file server holding the data (step 2), and waits for the reply. When the server replies with the data (step 3), the data is copied into the previously allocated page frame, and the kernel is invoked (step 4) to move or *migrate* the page frame to the faulting page address in the application's segment. The manager responds to the application, allowing it to resume (step 5). The figure assumes that the page data must be retrieved from the file server. If the manager has the page data available locally, steps 2 and 3 are replaced by an internal procedure in the manager which makes the data available in the page frame allocated to the application. With a copy-on-write fault the kernel performs the copy after the manager has allocated a page.

Filling the page frame tends to dominate the other costs of page fault handling because it usually requires either accessing backing store or copying from another page. Note that the kernel manages hardware-supported VM translation tables such as page tables and TLBs to map pages with the protections specified in the segment and bound region data structures. A page fault trap only occurs when a memory reference cannot be satisfied given the information in these kernel data structures. In particular, simple TLB misses are handled by the kernel.

The `MigratePages` operation operates on the page frames in bound regions by operating on the associated segments. For example, migrating a page frame to the address range corresponding to the data region in the virtual address segment in Figure 1 effectively migrates the page frame to the segment labeled *Data Segment*. Migrating a page frame to a segment is treated as a write operation for the purposes of segment protection and copy-on-write behavior. The `MigratePages` operation is also used to reclaim pages frames from segments as part of a page reclamation strategy.

Cached files, implemented as segments, can be accessed using a kernel-provided file-like block read/write interface, specifically the Uniform Input/Output Object (UIO) protocol [5]. A file read to a segment page that does not have an associated page frame causes a page fault event to be communicated to the manager of the segment, as for a regular page fault. File write operations requiring page allocation are handled similarly. File access performance is comparable to that of a system with a kernel-resident file system because, when the file is cached, the access is a single kernel operation [3] and when the file is not cached, the access time is dominated by secondary storage access costs.

The manager module can be executed by a process separate from the application or by the faulting process itself. In the first case, the kernel suspends the faulting process and communicates with the manager process using the interprocess communication facility. In the second case, the kernel transfers control to a procedure that is executed by the page faulting process, similar to a conventional signal or interrupt. This method is generally more efficient than the first method because no context switch is required. On some hardware, such as the MIPS R3000-based machines, resumption of the application after page faulting handling can be performed directly from the manager without going through the kernel, further improving the efficiency. Other systems, such as those using the MC 680X0 processors, require a return through the kernel to restore privileged pipeline state. With the potential of a variety of high-performance uses for application page cache management, we hope that future architectures will allow direct application resumption after fault, as in the R3000.

When the faulting process executes its own segment manager, some care is required in handling page faults on that process's stack to avoid infinitely recursive page faulting. Our approach is to use a separate fault-handling or signal stack that is always in memory, so a page fault in the page fault handling does not occur. There can be a separate signal stack per segment, so with a multi-threaded program, each thread can have a separate signal stack, namely one for its stack segment.

On initialization, the kernel creates a segment identified by a well-known segment identifier that includes all the page frames in the memory system, in order of physical address, with access limited to system processes, specifically the system page cache manager (see section 2.4). The system page cache manager uses the `MigratePages` operation to allocate these page frames to the various segment managers on demand. In a minimal configuration of the system, such as in an embedded real-time application with no demand paging, application processes can allocate pages directly from this initial segment, obviating the need for any process-level server mechanism. This scenario illustrates how the kernel virtual memory support contains very little extra mechanism beyond that required to support embedded applications, yet can be configured with process-level servers to implement a full demand-paging system with sophisticated application paging control.

In summary, the primary kernel extensions are: 1) the ability to designate an explicit manager for a segment, 2) kernel operations such as `MigratePages` and `ModifyPageFlags` to modify segments and page frame flags, and 3) the kernel operation `GetPageAttributes` to determine the page attributes for a range of pages frames. With the information and control exported by the kernel and the efficient communication to segment managers on page fault and page protection fault events, a process-level module can readily implement a variety of sophisticated schemes, including replicated writeback, page compression and logging, and it can coordinate writeback with the application, as is required for clean database transaction commit. In comparison to the external pager approach supported by the Mach kernel, the V++ kernel does no page reclamation and no page writeback.

[3] Using the block interface the file is not mapped into the address space of the reading process

2.2 Application-Specific Segment Managers

In each sophisticated large-scale application, an application-specific segment manager manages one or more of the application's memory segments. The management actions include: 1) handling page faults, 2) reclaiming pages from segments and 3) interacting with the system segment manager to allocate additional pages and return pages, as appropriate.

To handle page faults quickly, a segment manager typically maintains a *free-page segment*, just as is normally done by the kernel virtual memory system in a conventional design. The free-page segment is mapped into the manager's address space so the manager can directly copy data to and from the page frames as part of allocation and reclamation. For example, as part of a conventional page fault, it may read the page data from backing storage into a page in its address space that corresponds to the page in the free-page segment that has been allocated for this page fault. It then migrates the page frame to the faulting segment and allows the faulting process to continue.

More complex schemes are appropriate for some applications. For example, the segment manager for a database management system (DBMS) may use temporary index segments as free-page segments, and simply steal from these scratch areas rather than maintain explicit free areas. A DBMS segment manager may have a different free page segment for each of indices, views and relations, making it easier to track memory allocation to these different types of data. A single application may also use different segment manager modules for different segments or types of segments it uses. For example, it may maintain different free page segments to handle distributed physical memory on machines such as DASH [13] or for page coloring schemes. These techniques rely on being able to request page frames from the system page cache manager with specific physical addresses, or in particular physical address ranges.

The manager can implement standard page frame reclamation strategies, such as the various "clock" algorithms [12]. In particular, it can periodically migrate page frames from the segments it manages back to a free-page segment using `MigratePages`, keeping track of the segment and page number for each page frame it migrates, and writing back the dirty page data. If a given page frame is referenced through the original segment before the page frame is reused, the manager simply migrates it back to the original segment. The manager is also informed when a segment it manages is closed or deleted, so that it can reclaim the segment page frames at that time.

The manager can use application-specific strategies, such as deleting whole segments of temporary data that it knows are no longer needed or that are better to discard and regenerate in their entirety (rather than be paged out and back in, or regenerated a page at a time). Similarly, in a large-scale matrix computation, the manager may be able to prefetch pages of matrices to minimize the effect of disk latency on the computation while recognizing that it can simply discard dirty pages of some intermediate matrix rather than writing them back, thereby conserving I/O bandwidth.

On initialization, a segment manager requests the creation of its free-page segments with initial page frame allocations from the system page cache manager. It then creates further segments, possibly on demand from the application, to handle application data, specifying itself as the manager for these segments.

The issue of the page faults on segment manager code and data can be handled in two ways. First, the code and data can reside in segments that are managed by another manager, such as the default segment manager, described in the next section. Then, in the case of the first manager incurring a page fault on its code or data segment, this second fault is handled by the other segment manager before the first manager continues with the page fault handling. This approach is simple to implement, but does not provide predictable performance for the application segment manager. The alternative approach is for the application manager to manage the segments containing its code and data, and to ensure that these segments are not paged out while the program is active, effectively locking this portion in memory. In this approach, when an application starts execution, these segments are under the control of the default segment manager. The application manager accesses these pages at this point to force them into memory, then assumes management of these segments, and then reaccesses these segments, ensuring they are still in memory. A page fault after assuming ownership causes this initialization sequence to be retried until it succeeds[4] Once the manager has completed this initialization, it excludes its own page frames from being candidates for replacement. In this approach, to avoid all page faults in the page fault handling code itself, all segments must use a signal stack that is part of this effectively pinned data, not just the stack segments, as described earlier.

The same approach can be used when an application is swapped out to secondary storage. In particular, the application segment manager swaps the application segments except for its code and data segments. It then returns ownership of these latter segments to the default segment manager, and indicates it is ready to be swapped. The application manager is then suspended, and its segment pages are then swapped out as well. On resumption of the application, the manager gains control and repeats the initialization sequence described above.

An application segment manager can be "specialized" from a generic or standard segment manager using inheritance in an object-oriented implementation. The generic implementation provides data structures for managing the free page segment and basic page faulting handling. The page replacement selection routines and page fill routines can be easily specialized to particular application requirements. Thus, the application programmer's effort to provide page cache management is minimized, and focused on the application-specific policies and techniques, rather than the task of developing a segment manager from scratch.

2.3 Default Segment Manager

A default segment manager implements cache management for conventional programs, making them oblivious to external-page management. This manager executes as a server outside the kernel. In V++, the default segment manager is currently created as part of the "first team", a memory-resident set of systems servers started immediately after kernel initialization. Thus, the default manager does not itself page-fault.

In the V++ implementation, the UIO Cache Directory Server (UCDS) [5] has been extended to act as default segment manager. This server manages the V++ virtual memory system effectively as a file page cache. All address spaces are realized as bindings to open files, as in SunOS [5]. The original role of the UCDS was to handle file opens and closes so it could add files to the cache on demand and remove them as appropriate. In this original form, page faults were handled by the kernel once the mappings were established. The modifications for external page-cache management required extensions to this server to manage a free-page segment and to handle page fault requests, page reclamation and writeback. However, because it was already maintaining information about cached files on a per-file basis, the extensions to its data structures and overall functionality were relatively modest.

To determine the memory requirements of applications using the default segment manager, the default manager implements a clock algorithm [12] that allocates page frames to each requester based on the number of page frames it has referenced in some interval. The implementation of this algorithm requires passing a fault to the manager when a process first references a page after the page protection bits are set to disallow all references. The handling of the fault requires changing the protection of the referenced page. To reduce the overhead of handling these faults, the default manager changes the protection on a number of contiguous pages, rather than a single page, when a fault occurs. In general, the default manager can implement whatever algorithms that the corresponding kernel module would in a conventionally structured system, including page coloring and the like, if appropriate. Thus, the performance with the default segment manager should be competitive with conventional systems, as indicated by our measurements in Section 3.2.

2.4 System Page Cache Manager

The System Page Cache Manager (SPCM) is a process-level module that manages the allocation of the global memory pool among the segment managers. A manager requests some number of page frames from the SPCM in order to satisfy its memory requirements. The SPCM can grant, defer or refuse the request, based on the competing demands on the memory and memory allocation policy. The SPCM returns page frames to its local free page segment when returned by a segment manager, or when a segment manager terminates.

The SPCM can support segment manager requests for particular page frames by physical address or by physical

[4] This scheme assumes that the amount of memory required for the manager is small compared to the amount of physical memory. We do assume a large system memory configuration suitable for running the class of memory-bound applications motivating these techniques, where this assumption is invariably true. It is not clear that our approach is workable in general if the system memory resources are meager relative to the working set size of servers and other real-time or interactive modules.

[5] SunOS is a trademark of Sun MicroSystems Inc.

address range, as required for physical placement control and page coloring. If the SPCM cannot satisfy an allocation request because of physical address constraints, it is handled the same as a conventional (unconstrained) page frame request for which the size of memory requested is larger than that available. That is, it allocates and provides as many page frames as it can or is willing to. Further extensions can easily be provided for future architectures by modifying the SPCM, rather than complicating and destabilizing the kernel.

A "memory market" model of system memory allocation has been developed for the SPCM, and is explored in depth in a separate report [9]. In brief, the SPCM imposes a charge on a process for the memory that it uses over a given period of time in an artificial monetary unit we call a *dram*. That is, a process holding M megabytes of memory over T seconds is charged $M*D*T$ *drams*, if the charging rate is D drams per megabyte-second. A process is provided with an income of I drams per second of its existence, the value of I depending on the number of competing processes and the administration policy of allocating for the system. A segment manager as part of an application process thus must manage its dram supply to balance the cost of the memory used by the application versus its income. In particular, it must return memory to the SPCM when it can no longer afford to "pay" for the memory. The SPCM has the ability to force the return of memory from processes that have exhausted their dram supply, treating such process behavior as faulty.

For batch programs the application segment manager suspends and swaps the program until it has saved enough drams to afford enough memory for a reasonable time slice of execution. By queries to the SPCM, it can determine the demand on memory and possibly identify trade-offs between running in a small amount of memory soon versus waiting longer to get a larger amount of memory. When the process has enough drams to afford the memory, it requests the memory from the SPCM and runs as soon as the memory request is granted. At the end of its time slice, when its dram savings are running low, it pages out the data and returns to a quiescent state in which it has a very low memory requirement. As a further refinement, the SPCM can allow a process to continue to use memory at no charge when there are no outstanding memory requests. Also, there is a savings tax imposed to avoid demand dramatically exceeding supply, given this is basically a fixed price, fixed supply market. Finally, there is a charge for I/O that is based on the trade-off between memory and I/O in, for example, scan-structured programs, which prevents such programs from avoiding the memory charge with excessive I/O.

This monetary model allows the SPCM to allocate memory resources to programs according to the income supplied to each program, reflecting administrational policy. In particular, we claim that if each user account receives equal income, its programs also receive an equal share of the machine over time among the active users. This claim assumes a multiprocessor machine in which the primary limiting resources are memory and I/O. The monetary model also allows applications to decide how best to structure their computation relative to system resources, choosing for instance between computing with a large amount of memory for short timeslices versus computing for longer time slices with less memory. Finally, it provides a model that allows the segment managers to predict how long they can execute and the amount of memory available for that time. In the conventional approaches used for global page management developed during the 1960's and 70's, the application does not have any idea of when it might lose pages or be swapped. Moreover, implementing conventional working set algorithms would appear to either require trusting the application segment managers for information or largely duplicating their monitoring of the page access behavior.

Our results to date suggest that this approach results in a stable, efficient global memory allocation mechanism for large-scale computations that provides applications to considerable flexibility in making application-specific tradeoffs in the use of memory, thus matching well with the application control provided by the mechanisms described in this paper.

The V++ system page cache manager together with the default segment manager and the basic kernel virtual memory management provide the equivalent functionality of a conventional virtual memory system but in a more modular form. In particular, all the page I/O, replacement policies and allocation strategies have been moved outside the kernel. This is in line with our V++ objective of providing a minimal-sized kernel that is suitable for embedded real-time applications as well as conventional timesharing and interactive workstation use.

The small number of kernel extensions required for external page cache management could be added to a conventional Unix system, for example, to provide the benefits of application-controlled paging without the major surgery that would be required to revise the system design to match the modularity of V++. In particular, kernel extensions would be required to designate a mapped file as a page-cache file, meaning that page frames for the file would not be reclaimed (without sufficient notice), just as with the segments in V++. Also, a kernel operation, such as an extension to the `ioctl` system call, would be required to set the managing process associated with a given file and to allocate pages. (The kernel would be the default manager, as it effectively is now.) Finally, the `ptrace` and signal/wait mechanism can be used to communicate page faults to the process-level segment manager. The simplest solution to protecting the manager against page faults on its code and private data is simply to lock its pages in memory, a facility already available in Unix (although this may require the manager to run as a privileged process).

3 Evaluation

We have taken a two-pronged approach to evaluating external page-cache management. Firstly, we implemented external page-cache management in the V++ kernel and systems servers to work through the details of the design and evaluate its complexity and performance. Secondly, we evaluated the benefits of using external page cache management in a simulation of a database management system that uses a large amount of memory.

3.1 Measurements of System Primitives

External page-cache management was implemented in the V++ system by modification to the kernel virtual memory manager and extensions to the UCDS. In the kernel that uses external page-cache management, the machine independent virtual memory module is approximately 4500 lines of C code, as compared to approximately 6900 lines for the previous version. Most of the excised code is migrated to the page-cache managers so there is no real saving in the total amount of the code required for the same functionality. However it is significant in reducing the size of the kernel, (as well as providing greater external functionality).

The performance of the implementation was evaluated on a DECstation 5000/200. (R3000 processor with 25 MHz clock) which has a 4 kilobyte page size.

Table 1 summarizes the performance of V++ relative to ULTRIX 4.1.

Measurement	V++	Ultrix Equivalent
Faulting Process Minimal Fault	107	175
Default Segment Manager Minimal Fault	379	175
Read 4KB	222	211
Write 4KB	203	311

Table 1: System Primitive Times: times in microseconds

The minimal cost page fault (as measured in Table 1) occurs when the manager just has to migrate the page frame from its free page segment to the faulting process's segment. This case occurs frequently, such as on the first access to a heap page, on copy-on-write faults, and when write appending a new page to a segment.

The table measurements suggest that handling the minimal page fault is faster using the faulting process in V++ than through the Ultrix kernel. Most of the difference in cost (75 microseconds) is the cost of page zeroing that the Ultrix kernel performs on each page allocation. In Ultrix, zeroing is required for security because the page may be reallocated between applications, whereas this is not the case in V++ unless the page is being given to another user. Referring to the second row of the table, the cost of fault handling by the default manager is higher than in ULTRIX but this does not significantly affect the performance of applications as our measurements in the next subsection show.

Low overhead page fault handling allows efficient implementation of user level algorithms that use page protection hardware, like those described in [2]. Examples of these algorithms include mechanisms for concurrent garbage collection and concurrent checkpointing. In ULTRIX 4.1 on a DECstation 5000/200, the cost of a user level fault handler [6] for a protected page that simply changes the protection of the page is 152 microseconds. This is over 50% higher than the cost of handling a full fault using external page-cache management. ULTRIX is competitive at user level fault handling with other systems like Mach or SunOS. For example, in Appel and Li's measurements for the DECstation 3100 [2] the overhead of Mach fault handling operations was over twice the overhead of ULTRIX for similar operations.

The final measurements in the table are the costs of reading and writing a 4KB block in a cached file. In the case of V++ the accesses use the block read-write interface (discussed in Section 2.1). For ULTRIX we measured the cost of the *read* and *write* system calls. The V++ write cost is 34% less than ULTRIX. The V++ read cost is 5.2% higher than ULTRIX for reads. These numbers show that providing external page-cache management does not have a large negative effect on the performance of common operations like accesses to cached files.

3.2 Default Segment Manager

We ran a number of standard UNIX applications on V++ using the default segment manager with instrumentation to measure the overhead of executing real application programs using the default segment manager. For comparison we compiled the same source code (with different operating system dependent libraries) for ULTRIX 4.1.

The applications were:

1. diff: compare two 200KB files generating a differences file of 240KB.
2. uncompress: uncompress an 800KB file generating a file of 2MB.
3. latex: format a 100K input document generating a 23 page document.

In both cases the hardware was a DECstation 5000/200 with 128 megabytes of memory. The page size on this machine is 4KB. There are some differences between the two hardware configurations. The ULTRIX machine had a local disk. The V++ machine was diskless with file storage provided by a server running on a DECstation 3100 running ULTRIX 4.1.

These applications were run with the files they read cached in memory to eliminate differences in I/O performance that is irrelevant to the virtual memory system design factors we are measuring. These scenarios are also the worst-case for our approach because there is no network or file access latency to hide the cost of going to the V++ process-level manager.

There are some notable differences between V++ and ULTRIX. The unit of I/O transfer in ULTRIX is 8KB. The unit of I/O transfer in V++ is 4KB. This means that V++ makes twice as many read and write operations to the kernel as ULTRIX. Ultrix allocates pages in 4K units. The V++ default manager allocates pages in 4K units, except for appends to a file in which case it allocates pages in 16K units. The unit of page allocation is significant because allocation in V++ requires going to the segment manager. At the low levels of the virtual memory system, Ultrix uses page tables to describe address spaces. V++ augments the segment and bound region data structures with a global 64K entry direct mapped hash table with a 32 entry overflow area.

Table 2 shows the mean elapsed time for executing the programs under V++ and ULTRIX.

The measurements here show that the performance of applications in V++ is comparable to the performance of the same applications under ULTRIX.

To attempt to account specifically for the differences in performance, we also measured the virtual memory system activity of each progam, as shown in Table 3.

[6] In ULTRIX a user-level fault handler can be implemented using a signal handler and the `mprotect` system call, which changes the protections of an application program's memory.

Program	V++	Ultrix
diff	3.99	4.05
uncompress	6.39	6.01
latex	14.71	13.65

Table 2: Application Elapsed Time in Seconds

Program	Manager Calls	Migrate Pages	Manager Overhead
diff	379	372	76 mS
uncompress	197	195	40 mS
latex	250	238	51 mS

Table 3: VM System Activity and Costs

Column 1 shows the number of times during the execution of the program that the manager was invoked, including requests forwarded by the kernel for operations like closing a file as well as requests for a page frame. Column 2 shows the number of times the manager invoked `MigratePages`. This column basically shows that almost all manager calls were to handle page faults rather segment releases or other management operations. Column 3 shows the cost in milliseconds of using the V++ manager, calculated as the difference in cost between a minimal page fault to the default segment manager in V++ and the corresponding cost in Ultrix (from Table 1) multiplied by the number of manager calls.

The cost of the V++ process-level handling of page faults is a small percentage of program execution time even for the measured case where there is no disk or network access (1.9% for diff, 0.63% for uncompress and 0.35% for latex).

The differences in application performance between V++ and Ultrix in Table 2 not accounted for by Table 3 we attribute to differences in the run-time library implementations in V++ and Ultrix. Of the applications measured, only `latex` under V++ is significantly slower, and we are continuing to investigate the reason. However, our measurements in Table 3 indicate that the external page cache management is not responsible for more than 51 milliseconds or about 4.8% of the difference in execution times.

Overall, assuming that the applications we have measured are representative of those to be run under the default segment manager in V++, we conclude that minimizing the kernel using external page cache management does not introduce a significant overhead on normal programs. In fact, we expect that the V++ overhead suggested by the measurements has been somewhat overstated because a system under normal conditions would have a significant number of page faults that include disk or network I/O, whereas we have eliminated these costs in the measurements to provide a worst-case for V++ and to avoid spurious differences arising from device behavior.

3.3 Application-Specific Page-Cache Management

To explore the performance benefits of application-specific page-cache management we developed a program that simulates a database transaction processing system that exploits a space-time tradeoff in its use of indices for efficient join processing. If memory is plentiful, it is more efficient to perform large joins by generating indices for the relations in advance. If however, the creation and references to the indices would result in additional paging, it is better to discard indices for which there is not enough space, and regenerate them in memory when they are needed.

The program was run using 6 processors of a Silicon Graphics 4/380 on a 120 megabyte database. The transaction arrival rate was 40 transactions per second, The transaction mix was 95% small DebitCredit type transactions with the remaining 5% being joins of two relations to update a third. A hierarchical locking scheme is used for concurrency control.

The program is a mixture of implementation and simulation. The locks were implemented and the parallelism is real. However, the execution of a transaction is simulated by looping for some number of instructions and a page fault is simulated by a delay that is equivalent to the time required to handle a page fault on the SGI 4/380.

The measurements in Table 4 show the performance differences between four configurations of the database program.

Configuration	Average Response	Worst-case Response
No index	866	3770
Index in memory	43	410
Index with paging	575	3930
Index regeneration	55	680

Table 4: Effect of Memory Usage on Transaction Response (ms)

The first configuration shows the response time when no index is used for joins. The second configuration shows the reduction in response time achieved by using an index for accessing relations for performing a join, in the case where the indices are always in memory. In the case of the configuration labeled "index with paging", a one megabyte index is paged in every 500 transactions (on average every 12.5 seconds) because the size of the virtual memory used by the program exceeds the memory allocated to the program by 1 megabyte.

These measurements show that indices are of significant benefit to response time if the (physical) memory is available, but are of limited benefit if the size of the database system's virtual memory exceeds the available physical memory by less than 1% and there is a modest amount of paging.

If the database system is informed that its virtual memory size exceeds the physical memory allocated to it, it can discard some indices and regenerate them when necessary. The "index regeneration" entry shows the performance benefits of this approach after the physical memory allocated to the database system is reduced by 1 megabyte. In this case, the average response time is an order of magnitude less than the paging case and is only 27% worse than the "index in memory" case.

This example demonstrates a case of application-controlled page cache management having significant benefits even though the application's virtual memory only slightly exceeded available physical memory. We expect similar benefits with other memory-intensive applications.

4 Related Work

The inadequacy of the conventional "transparent" virtual memory model is apparent in recent developments and papers in several areas. For example, Hagmann [11] proposed that the operating system has become the wrong place for making decisions about memory management. He discussed the problems with current VM systems, but did not present a design that addresses these problems.

The conventional approach of pinning pages in memory does not provide the application with complete information on the pages it has in memory because the application typically does not, and cannot, pin all the pages it has in memory. The operating system cannot allow a significant percentage of its page frame pool to be pinned without compromising its ability to share this resource among applications. The amount of pinning that is feasible is dependent on the availability of physical memory. These complications have led many systems, particularly different versions of Unix, to restrict memory pinning to privileged systems processes or to impose severe limits on the number of pages that can be pinned by a process. The extension of pinning to "notification" locks, so a process is notifiable when a pinned page is to be reclaimed, would allow more pinning but would still not give the application control over which page frames can be reclaimed. With external page cache management in V++, the system page cache manager can reclaim page frames from applications, but the application's segment manager(s) have complete control over which page frames to surrender. We expect that, with the appropriate generic segment manager software, developing an application-specific segment manager should be no harder than developing a "pin" manager module. However, further experience is required in this area before firmer conclusions can be drawn.

The *external pagers* in Mach [21] and V [5] provide the ability to implement application-specific read-ahead and writeback using backing servers or external pages. However, these extensions do not address application control of the page cache and are primarily focused on the handling of backing storage. The PREMO extensions to Mach [14] address some of the shortcomings of Mach noted in Young's thesis [22]. PREMO supports user-level page replacement policies. The PREMO implementation involves adding more mechanism to the Mach kernel, to deal with one aspect of the page-cache management problem – page replacement, thus complicating rather than simplifying the kernel, as we have done. PREMO also does not export information to the application level about how much memory is allocated to a particular program.

In [18] Subramanian describes a Mach external pager that takes account of dirty pages that do not need to be written back. She shows significant performance improvements for a number of ML programs by exploiting the fact that garbage pages can be discarded without writeback. She proposes adding support to the kernel for discardable pages to remedy two problems associated with supporting discardable pages outside the Mach kernel. First, an external pager does not have knowledge of physical memory availability. Second, there are unnecessary zero-fills (for security) when a page is reallocated to the same application. Both of these problems are addressed by external page-cache management without adding special mechanism to the kernel.

Database management systems have demanded, and operating systems have provided, facilities for pinning pages (such as the Unix `mpin` and `mlock` calls) and limited advisory capability, such as the Berkeley Unix `madvise` call. However, these approaches provide simple ways to prevent page out or to influence paging behavior, not a real measure of control of the page cache by a program, as we have proposed. Support for application-designated page replacement on a per-page basis and notification of changes in available physical memory are well beyond the scope of the design, as well as the implementation, of these current facilities.

Discontent with current virtual memory system functionality is evident in the database literature, both in complaints about the virtual memory system compromising database performance, and in the calls for extended virtual memory facilities [16, 19] or the elimination of the virtual memory system altogether. We see our approach as providing the database management systems with the information and control of page management demanded in this literature. We achieve this without compromising the integrity of the operating system or its general purpose functionality.

This work has some analogy to proposed operating system support for parallel application management of processors. For example, Tucker and Gupta [20] show significant improvements in simultaneous parallel application execution if the applications are informed of changes in the numbers of available processors and thereby allowed to adapt, as compared to the conventional transparent, oblivious approach. Anderson et al. [1] and Black [4] have proposed kernel mechanisms for exporting more control of processor management to applications. Just as in our work, this processor-focused work is targeted to the demanding applications whose requirements exceed what are, by normal standards, plentiful hardware resources. Both our work and the processor-focused work are not targeted towards improving the performance of conventional applications such as software development tools and utilities. Our work complements this other work by focusing on application control of physical memory, rather than control of processor allocation.

5 Concluding Remarks

External page-cache management is a promising approach to address the demands of memory-bound applications, providing them with control over their portion of the system memory resources without significantly complicating system facilities, particularly the kernel.

We have argued that the cost of a page fault is too high to be hidden from the application, except for its effect on performance. Our measurements of a simple simulated parallel database transaction processing application support this view, showing that a small amount of paging can eliminate any performance benefit of algorithms that use virtual address space just slightly in excessive of the amount of physical memory available, compared to those more economical in space. This behavior is consistent with memory thrashing behavior we have observed with memory-bound applications in general. It is strange that, while the space-time tradeoff is well-recognized by the algorithms community and a choice of algorithms exists for many problems that offer precisely this tradeoff, virtual memory systems have not previously exported the information and control to the applications to allow them to make the choice of algorithm intelligently. With the cost of a page fault to disk in the hundreds of thousands of instructions for the foreseeable future, an application can only expect to trade space for time if the space is real, not virtual.

External page-cache management, as implemented in the V++ system, requires relatively simple extensions to the kernel, and provides performance for user page fault handling times that are less than 110 microseconds on current conventional hardware. Our approach also subsumes the external pager mechanism of Mach and V. External page-cache management obviates the need to provide kernel support for the various application-specific advisory and monitoring modules that would otherwise be required in the future, causing a significant increase in kernel code and complexity. That is, we argue that the complexity and code size benefits are best appreciated by considering the size and complexity of a Unix `madvise` module that could deal with the memory management problems raised in this paper. In that vein, we expect that other considerations, such as page coloring, physical placement control and and cache line software control, as in ParaDiGM [8], to place further demands on memory management software in the future.

Finally, we have exploited the new external page cache management kernel operations to further reduce the size of the V++ kernel by implementing system page cache management and a default segment manager outside the kernel. These changes have lead to a significantly simplified kernel, because page reclamation, most copy-on-write support and distributed consistency are all removed to process-level managers.

A primary focus of our on-going work is on the development of application-specific segment managers, based on a generic manager, using object-oriented techniques to specialize this infrastructure to particular application requirements. The goal is to minimize the burden on application programs while providing the benefits of application control. We are also investigating the market model of system-wide memory management and its performance in sharing system memory resources between competing applications. With our primary focus on batch processing, results to date have been promising.

The external page cache management approach develops further a principle of operating system design we call *efficient completeness*, described previously in the context of supporting emulation [6]. The operating system kernel, in providing an abstraction of hardware resources, should provide efficient and complete access to the functionality and performance of the hardware. In the context of memory management, the complete and efficient abstraction of this hardware resource is that of a page-cache. Fair multiplexing of memory among the multiple competing applications is achieved by managing the page frame allocation among these page caches. It also generally leads to a relatively low-level service interface, thereby being in concert with the goals of minimalist kernel design, as we have shown with external page cache management.

In summary, we believe that external page-cache management is a good technique for structuring the next generation of kernel virtual memory systems, addressing the growing complexity of memory system organizations and the growing demands of applications while reducing the size of kernel virtual memory support over conventionally structured systems. Once this facility is commonly available in commercial systems, we expect the most exciting memory management improvements may well come from the developers of database management systems, large-scale computations and other demanding applications whose performance is currently badly hindered by the haphazard behavior of conventional virtual memory management.

6 Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211. The comments of the ASPLOS referees have helped to improve the quality of the paper. We thank Anita Borg, Mendel Rosenblum, John Chapin and Hendrik Goosen for their comments and criticisms. We are extremely grateful to Jonathan Stone for the many hours he spent helping perform the measurements for this paper.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy
Scheduler Activations; Effective Kernel Support for the User-Level Management of Parallelism
ACM Transactions on Computer Systems, 10(1), February 1992.

[2] Andrew Appel and Kai Li
Virtual Memory Primitives for User Programs
In Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991.

[3] A. Bensoussan, C. T. Clingen and R. C. Daley
The Multics Virtual Memory
In Proceedings of the 2nd ACM Symposium on Operating Systems Principles, Princeton, New Jersey, October 1969.

[4] David Black
Scheduler Support for Concurrency and Parallelism in the Mach Operating System
IEEE Computer Magazine, 23(5):35-43, May 1990.

[5] David R. Cheriton
The V Distributed System
Communications of the ACM, 31(3):314-333, March 1988.

[6] David R. Cheriton, Gregory R. Whitehead and Edward W. Sznyter
Binary Emulation of Unix using the V Kernel
Usenix Summer Conference, June, 1990.

[7] David R. Cheriton, Hendrik A. Goosen and Philip Machanick
Restructuring a Parallel Simulation to Improve Shared Memory Multiprocessor Cache Behavior: A First Experience
Shared Memory Multiprocessor Symposium, Tokyo, Japan, April 1991.

[8] David R. Cheriton, Hendrik A. Goosen and Patrick D. Boyle
ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture
IEEE Computer
24 (2), February, 1991.

[9] David R. Cheriton
A Market Approach to Operating System Memory Allocation
Working Paper, March 1992.

[10] Daniel Dobberpuhl et al.
A 200MHz 64b Dual-Issue CMOS Microprocessor
In the 39th International Solid-State Circuits Conference, pages 106-107, February 1992.

[11] Robert Hagmann
Comments on Workstation Operating Systems and Virtual Memory
In Proceedings of 2nd IEEE Workshop on Workstation Operating Systems, Pacific Grove, California, September 1989.

[12] Samuel Leffler et al.
The Design and Implementation of the 4.3 BSD UNIX Operating System
Addison-Wesley, November 1989.

[13] Dan Lenoski et al.
The DASH prototype: Implementation and Performance
In Proceedings of 19th Symposium on Computer Architecture, pages 92-105, May 1992

[14] Dylan McNamee and Katherine Armstrong
Extending the Mach External Pager Interface to Allow User-Level Page Replacement Policies
Technical Report 90-09-05, University of Washington, September 1990.

[15] Brian K. Bray, William L. Lynch and M. J. Flynn
Page Allocation to Reduce Access Time of Physical Caches
Technical Report CSL-TR-90-454, Computer Systems Laboratory, Stanford University, Novemeber 1990.

[16] Michael Stonebraker
Operating System Support for Database Management
Communications of the ACM, 24(7):412-418, July 1981.

[17] Michael Stonebraker et al.
The Design of XPRS
Memorandum No. UCB/ERL M88/19, University of California Berkeley, March 1988.

[18] Indira Subramanian
Managing Discardable Pages with an External Pager
In Proceedings of the Second Usenix Mach Symposium, Monterey, California, November 1991.

[19] Irving Traiger
Virtual Memory Management for Database Systems
Operating Systems Review, 16(4):26-48, April 1982.

[20] Andrew Tucker and Anoop Gupta
Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors
In Proceedings of 12th ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona, December 1989.

[21] Michael Young et al.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System
In Proceedings of 11th ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987.

[22] Michael W. Young
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System
PhD thesis, Department of Computer Science, Carnegie Mellon University, November 1989. Also available as Technical Report CMU-CS-89-202.