October 1991 Report No. STAN-CS-91-1394

Application-Controlled Physical Memory Using External Page-Cache Management

by

Kieran Harty and David R. Cheriton

Department of Computer Science

Stanford University

Stanford, California 94305

Application-Controlled Physical Memory using External Page-Cache Management

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

Abstract

Next generation computer systems will have gigabytes of physical memory and processors in the 100 MIPS range or higher. Contrary to some conjectures, this trend requires more sophisticated memory management support for memory-bound computations such as scientific simulations and systems such as large-scale database systems, even though memory management for most programs will be less of a concern.

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 physical page cache provided by the kernel. 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 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. With the increasing speed of processors and the lack of comparable improvement in I/O latency or throughput making the memory system the key performance limiting factor for these demanding applications, the "instruction budget" exists to take a more intelligent approach to physical page management.

There are three major problems with current virtual memory systems. First, an application cannot know the amount of physical memory it has available, and it is not informed when significant changes are made in the amount of available memory. A second deficiency is that 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.

If an application has information about the availability of physical memory, it can make an intelligent space-time tradeoff between different algorithms or modes of execution that achieve its desired computation. (Clearly, making that tradeoff using virtual memory that is not backed by physical memory fails to take into account the time cost of paging, which can easily dominate in many computations.) For example, we have studied a large scale particle simulation [8] based on the Monte-Carlo method that generates a final result based on the averaging of a number of runs of the simulation. The number of runs required to generate a statistically accurate result can be reduced (without a proportionate increase in computation time) by increasing the number of particles (and the memory requirements). It is possible to combine the results of runs using different numbers of particles, so that the number of particles per run does not have to be fixed before beginning the first run of the simulation. If the program is informed of the amount of space that is available, it can modify the number of particles to be used for runs of the simulation. It is also possible to discard the state associated with a particular run during its execution, so that aggressive use of space by the simulation does not cause other applications to wait a long time to reclaim memory. This adaption to the amount of available memory is also important for parallel database query processing [14] in which the amount of memory used is dependent on the amount of parallelism, and in which the amount of parallelism used should be dependent on the available physical memory. Another example is an application using garbage collection running in virtual memory. When physical memory is plentiful, it may use a large amount of virtual memory to minimize the frequency of garbage collections. However, garbage collection across a virtual address space whose size significantly exceeds the available physical memory leads to heavy paging.

If an application can control the portion of its virtual address space contained in physical memory, it may be able to operate far more efficiently, even though it is using a virtual address space that far exceeds the size of physical memory. For example, a database management system should be able to ensure that critical pages, such as those containing central indices and directories, are in physical memory as well as be able to provide complete information to the query optimizer and transaction scheduler on which pages it has in memory. The cost of a single extra page fault, as arises from a conventional random page-out by the operating system, could significantly affect the overall cost of a typical query and dramatically extend the lock hold time if locks are held across a page fault. With multiprocessor machines, an unfortunate page fault can cost not just the elapsed time of hundreds of thousand instructions, but that cost multiplied by the number of processes blocked if they also hit the same page or a lock held by the blocked process.

The conventional approach of pinning pages in memory does not provide the application with complete information on the pages it has in memory. Pinning also makes applications more complex and harder to maintain. [1] In particular, the application must remember to pin and unpin pages based on changing program activity, even though most of the pages are never candidates for replacement, at least in the large memory systems of interest. The pinning approach is also limited by the necessary limits that the virtual memory system must place on the number of pages which can be pinned at any one time. In particular, the operating system cannot allow a significant percentage of its physical page 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

[1]The **problems are similar to those associated with writing programs using overlays in a previous generation of systems.**

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.

If an application can control the read-ahead, writeback and discarding of pages within its physical memory, it can efficiently schedule these activities to minimize the effect of I/O latencies on its execution and minimize I/O bandwidth requirements. For example, large-scale scientific computations using matrices 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 write-behind is supported by the operating system (and the requisite I/O bandwidth is available). For example, the large-scale parallel particle simulation cited above scans the entire data set of 200 megabytes for each simulated time interval, and takes approximately 12 seconds to do so on a machine with 8 30 MIPS processors [2] if the entire data set is in physical memory. The total scan time provides a significant amount of time to hide prefetching and writeback if the data does not fit entirely in memory, if the operating system supports the appropriate functionality.

The ***external pagers*** in Mach [18] and V [6] provide the ability to implement application-specific read-ahead and write-behind. However, they do not provide application control of physical memory for the first two problems. Both of these systems also retain a significant amount of mechanism in the kernel which we shall show can be migrated to the process level.

In this paper, we describe the design, implementation and evaluation of a virtual memory system that provides the required application control of physical memory using what ***we call external page-cache management.*** With external page-cache management, the virtual memory system provides the application with one or more physical page-caches which the application can manage external to the kernel. In particular, it can know the exact size of the cache in physical pages, 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, regeneration and write-behind. In essence, a key novelty of our design is that the virtual memory system provides a physical page 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.***

This approach allows significant improvements in execution for memory-bound programs along the lines described above, without compromising the simplicity or execution efficiency of conventional programs. It also leads to a simpler and smaller kernel. The reduction in kernel complexity is particularly pronounced if weighed against the expected kernel complexity of supporting virtual memory extensions to handle the application requirements described above. 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 prototype database transaction processing system. Section 4 describes related work. We close with a discussion of our future plans and conclusions.

[2] Silicon Graphics 4D/380

2 External Page-Cache Management in V++

External page-cache management entails providing kernel operations to allow process-level management of physical pages and process-level page-cache manager modules. This section describes an implementation of external page-cache management in V++, a new generation of the V distributed system.

2.1 Kernel VM Support

A ***segment,*** the main kernel-supported VM object, is a variable-size sequence of zero or more pages, similar to the conventional virtual memory notion of segment [5]. Pages can be added, removed, mapped, unmapped, read and written using segment operations. V++ provides several extensions to support external (to the kernel) management of segments.

An explicit ***page cache manager*** is associated with each segment that is responsible for managing the pages associated with the segment. In particular, when a reference is made to a missing or protected page in a segment, the event is communicated to the page-cache manager which is expected to handle the fault following the sequence illustrated in Figure 1. In the case of a missing

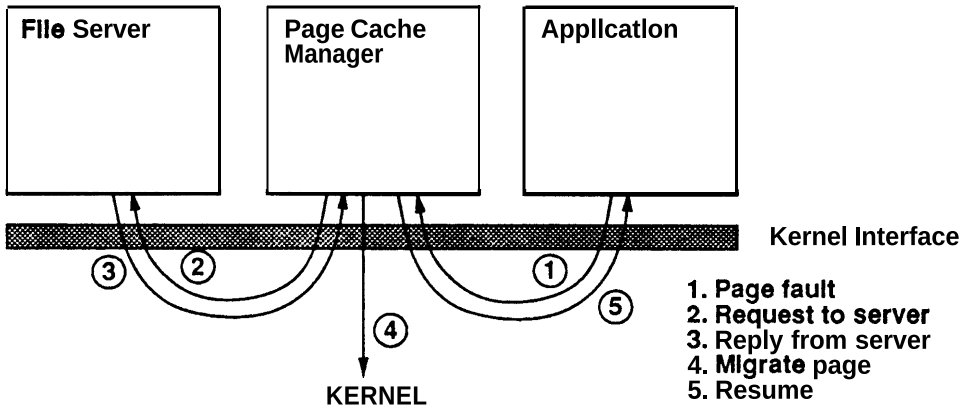

Figure 1: Page Fault Handling with External Page-Cache Management

page, the page-cache manager allocates a physical page frame from another segment (often a ***free-page segment),*** fills it with the appropriate data and migrates the page to the segment using the kernel `MigratePages` operation. `MigratePages` moves a set of physical page frames starting from a designated offset in one segment to another, optionally revising the flags, such as the ***dirty flag,*** associated with the affected page frames. The manager may map portions of the segment into its address space to read and write the associated page frames using the standard VM memory mapping support. In particular, to satisfy a conventional page fault, it may read the page data from backing storage into a page associated with a free-page segment which is mapped into its address space. It then migrates the page to the faulting segment and allows the faulting process to continue. In the case of a copy-on-write fault, the page-cache manager allocates a page, copies the contents of the protected page to the new page, and migrates the page (flagged as writable) to the referencing segment, and allows the referencing process to continue. Thus, the cost of a page fault consists of the costs of: 1) trapping to the manager, 2) filling the page, 3) migrating the page and

4) restarting the faulting process. Filling the page usually requires either accessing backing store or copying from another page, so this cost tends to dominate the other costs.

The fault handler can be executed by the faulting process itself or as a separate external page-cache manager process [3]. In the case where the fault is being handled by the faulting process, the kernel transfers control to a procedure that is executed by the faulting process. The procedure handles the fault (this may require communication with other servers such as a file server), and restores the previous context of the faulting process. When the page-cache manager is a separate process, the kernel suspends the faulting process until the manager has handled the fault and requested restart.

The **MigratePages** operation is used to reclaim pages from segments as part of a page reclamation strategy. The kernel also provides operations to access information about the amount of physical memory that is available, query and modify the state of the pages, including page flags like dirty bits, and query/modify the page-cache managers of segments. The use of these operations is described more fully in the next section.

Segments are used to implement cached files using a kernel-provided file-like block read/write interface, specifically the Uniform Input/Output Object (UIO) protocol [4]. In particular, applications can directly read from, and write to, segments created by a file cache manager. A file read to a segment page which does not have an associated physical 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. This extension allows file access performance that 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 in both cases, and when the file is not cached, the access time is usually dominated by secondary storage access costs.

A virtual address space in V++ is implemented as a segment, is composed from one or more segments. This composition is achieved by binding one or more ***regions*** of one segment into another segment. A bound region 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. A reference to an address covered by a bound region in one 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. On write to such a page, the binding is changed to refer to that in the destination segment after the page-cache manager has handled the necessary page allocation and copying, as described above. A **MigratePages** operation operates on the pages in bound regions by operating on the associated segments. Migrating a page to a segment is treated as a write operation for the purposes of segment protection and copy-on-write behavior.

A virtual address space with two bound regions to two separate segments is illustrated in Figure 2. The kernel manipulates 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.

In comparison to the external pager approach supported by the Mach kernel, the V++ kernel does no page reclamation and no page writeback. In fact, its functionality matches that required in a real-time embedded system with no demand paging, but wanting to use virtual address spaces.

[3] We use the term page-cache *manager* to refer to a module outside the kernel managing segments unless otherwise noted. The only kernel module to serve as a page cache manager, the kernel memory server, is a page-cache manager of last resort.

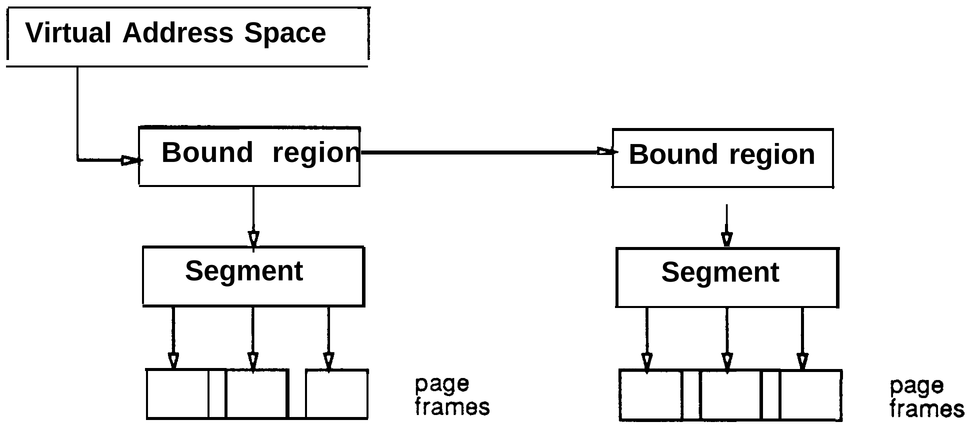

Figure 2: ‘Kernel Implementation of a Virtual Address Space

In this case, the kernel facilities would serve to manage physical memory and catch erroneous references to incorrect virtual addresses. With the information and control exported by the kernel and the efficient communication to page-cache managers on page fault and page protection fault events, process-level modules 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.

Application-specific page management techniques are discussed in the next subsection. The following subsection describes the system ***page-cache manager*** which manages page allocation to application-specific page-cache managers.

2.2 Application-Specific Page-Cache Managers

An application-specific page-cache manager is responsible for allocation of physical pages to the application segments it manages. To ensure that pages are available to handle page faults, it can maintain a free-page segment to serve as a holding area for available pages, The free page segment is similar in function to the free page list in most virtual memory systems but it is specific to a particular page-cache manager. More complex schemes are appropriate for some applications. For example, the page-cache 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.

Because a segment is implemented as essentially an open file, and the cost per open file is small, the system cost of having multiple free-page segments is minimal. The application is more limited by the complications and fragmentation resulting from the use of multiple segments. For example, a DBMS page-cache manager might have different page-caches for indices, views and relations, making it easier to track memory allocation to these different types of data, but may not find having a segment per index effective.

The page-cache manager can implement standard page reclamation strategies, such as the various “clock” algorithms [11]. In particular, it can periodically migrate pages from the segments it manages back to a free-page segment using `MigratePages`, keeping track of the segment and page number for each page it migrates, and writing back the dirty pages. If a given page is referenced through the original segment before the page is reused, the page-cache manager simply migrates it back to the original segment. However, it can also use application-specific strategies, such as deleting whole segments of temporary data that it knows are no longer needed or that it is better to discard and regenerate in their entirety (rather than be paged out and back in, or regenerated a page at a time). The page-cache manager is informed when a segment is closed or deleted, so that it can reclaim the segment pages at that time.

An application-specific page-cache manager can implement application-specific read-ahead, write-behind, page discard and regeneration policies and techniques. For example, as part of a large-scale matrix computation, the page-cache 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 page-cache manager requests the creation of its free-page segments with initial page allocations from the system page-cache manager. It then creates further segments, possibly on demand from the application, to handle application data, specifying the managing segment for these segments to be one of the free-page segments. The manager itself is associated with the management of these segments as ***owner*** of the managing segment. This server can reside in the same address space as the application(s) it manages or in a different address space.

A page-cache manager needs to ensure that its own code and data pages are not subject to random replacement by the rest of the system in order to avoid deadlock. To this end, there is an operation to change the managing segment for a specified segment, **WriteManagingSegment**. On initialization, the page-cache manager can set itself as the manager of the segments representing its own code and data. Recursive page faulting in the manager is avoided by "pretouching" the critical pages immediately before and after assuming ownership, and retrying the initialization if this fails.

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

In the V++ implementation, the UIO cache directory server (UCDS) [6] has been extended to act as default page-cache manager. This server manages the V-l-+ virtual memory system effectively as a file page cache. All address spaces are realized as bindings to open files, as in SunOS [4]. 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 default page-cache management, the default page-cache manager implements a clock algorithm [11] that allocates pages to each requestor based on the number of pages it has referenced in some interval. The implementation of this algorithm requires passing a fault to the page-cache manager when a process first references

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

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 page-cache manager can change the protection on a number of contiguous pages, rather than a single page, when a fault occurs.

2.3 System Page-Cache Manager

The System Page-Cache Manager (SPCM) is a process-level server that implements a global policy for allocation of memory among page-cache managers, including the default page-cache manager. The SPCM is the only process-level component permitted to allocate and free physical memory. If another process-level component invokes a kernel operation to allocate physical memory to a segment, the request is forwarded to the SPCM. If the SPCM finds the request acceptable, it allocates physical memory by migrating pages from a special kernel segment. To free physical memory associated with a segment, the SPCM migrates the pages to the special kernel segment.

An application-specific page-cache manager can request additional physical pages from the SPCM, the response depending on the availability of memory. A page-cache manager can also request information from the SPCM about overall memory usage. A page-cache manager must be prepared to respond to requests from the SPCM to release physical pages, which may arise in response to increased demand from other page-cache managers. The application-specific page-cache manager is responsible for freeing pages back to the SPCM in a timely fashion. It is also responsible for performing any necessary writebacks. The SPCM has the ability to unilaterally reclaim pages using the `MigratePages` operation, although this option is only intended to handle malfunctioning page-cache managers.

The SPCM allocates memory to page-cache managers based on its policy and on information provided by the page-cache managers. A simple allocation policy is to slice the memory equally among memory requestors if all requests cannot be satisfied. If a page-cache manager requests less than its slice, the excess memory is divided fairly among those requestors that have requested more than their slice. Periodically, the SPCM requests each page-cache manager to provide information about its memory requirements. The SPCM does not try to assess the requirements of page-cache managers by monitoring reference patterns. To discourage wasteful usage of memory when memory requirements exceed memory availability, the SPCM implements a cost model that encourages memory users to relinquish memory when aggregate demand approaches supply.

The SPCM also requests information about the number of pages allocated to a particular page-cache manager that require writeback. This information is used by the SPCM to reduce the delay associated with reallocation of physical memory from one page-cache manager to another. Estimating the delay is especially significant when the I/O bandwidth available for writeback is low. The number of pages requiring writeback is not necessarily the same as the number of pages allocated to a page cache manager that have their dirty bit set, For example, temporary objects and the uncompressed form of a compressed object that has not been modified, do not require writeback, even though the pages associated with these objects may be dirty. Based on the data from page-cache managers and its policy on delay, the SPCM may tell a page-cache manager to keep some number of its pages clean. If the SPCM expects to meet its delay targets with the current number of pages that require writeback, it does not ask the page-cache managers to clean any pages.

For brevity, the above description omits many details of the virtual memory system that are similar to conventional memory systems, or have been described previously [6], but does cover the essential aspects of the design.

3 Evaluation

We have taken a two-pronged approach to evaluating external page-cache management. First, we implemented external page-cache management in the V++ kernel and systems servers to investigate the feasibility of the design and its complexity implications. We measured the performance of the implementation. Second, we evaluated the benefits of using external page cache management in a prototype 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. These totals include comments, blank lines and headers with function prototypes. 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. We view it as 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 (R3000 processor with 33 MHz clock) which has a 4 kilobyte page size. The cost of a page fault that is handled by the faulting process without communicating with other servers is 97 microseconds.

This page fault handling cost is less than in current commercial systems. For example, the cost of page fault handling by the application is significantly lower than the kernel cost of handling a page fault to newly allocated data in ULTRIX 4.1 on identical hardware. In ULTRIX, this cost is 175 microseconds, which is dominated by the overhead of automatic zeroing of the page. Note that this zeroing is required for security because the page may be reallocated between applications, whereas this is not the case with application-controlled page cache management unless the page is being given to a new page cache manager.

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, the cost of a user level fault handler [5] for a protected page which simply changes the protection of the page is 152 microseconds. This is over 50% higher than the cost of handling a full fault (which requires more real work) using external page-cache management. It is worth noting that ULTRIX is quite efficient at user level fault handling, relative to 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.

[5]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.

3.2 Application-Specific Page-cache Performance

To explore the performance benefits of application-specific page-cache management we developed a prototype of a database transaction processing system that exploits a space-time tradeoff in its use of indices for efficient join processing. In essence, 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 prototype 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 prototype 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 1 show the performance differences between four configurations of the database prototype.

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 1: Effect of Memory Usage on Transaction Response Times (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 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. Hagmann [10] 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.

Both Mach and V extend the virtual memory system to allow the use of application-specific backing servers or external pagers. 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 [12] address some of the shortcomings of Mach noted in Young's thesis [19]. 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 rat her 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 a recent paper [15] 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. Specifically, the current `madvise` system call only provides hints for sequential readahead and page reclamation for mapped files. 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 [13, 16] 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, as some of the proposals would, and without compromising the general-purpose functionality of the system, as would result from "throwing out" the virtual memory system altogether, as others have advocated.

This work has some analogy to proposed operating system support for parallel application management of processors. For example, Tucker and Gupta [17] 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 [3] 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 historical standards, plentiful hardware resources yet is largely irrelevant to many 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 Future Extensions

A generic page-cache manager library is needed to minimize the cost and complexity of developing new application-specific page-cache managers, We are using object-oriented programming techniques to provide a basic framework for page-cache management while allowing extensibility to meet the requirements of specific applications.

It seems feasible to add the functionality of external page-cache management to the Unix kernel without major restructuring. 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 page-cache segments in V++. Second, a kernel operation, such as an extension to the `ioctl` system call, would be required to set the managing page-cache open file associated with a given file and to allocate pages. (The kernel would be the default page-cache manager, as it effectively is now.) Finally, the `ptrace` and signal/wait mechanism can be used to communicate page faults to the process-level page-cache manager. The simplest solution to protecting the page-cache manager against page faults on its code and private data is to simply lock its pages in memory, a facility already available in Unix (although this may require the page-cache manager to run as a privileged process). With these simple modifications, it seems feasible to make many of the benefits of external page management available in Unix (the exception being the reduction in kernel size in V++). The programming of page-cache management would be a small percentage of the cost and complexity of application systems such as database management systems and large-scale computations which motivate these extensions, and the performance benefits would be significant.

Finally, additional work is needed and planned for implementing cooperative memory management between the system page-cache manager and application-specific page-cache managers under system conditions that normally cause swapping [6]. In a conventional system, the memory manager must discover that an application's effective working set is larger than the amount of memory it can be allocated under the current load conditions. With application-specific page-cache managers, the application can adapt to less memory in some cases and otherwise indicate the amount of memory it requires to run efficiently. With the system page-cache manager implementing a cost-model for memory and processing, the application-specific page-cache managers can optimize for throughput and response, as appropriate for their application, while the system page-cache manager optimizes system performance overall. While this work has yet to be done, we expect that many established virtual memory techniques will apply and that radically new techniques are unlikely to be required in this domain.

[6]By swapping a program (in a virtual memory system), we mean that a program is basically removed from consideration for execution so it stops contending for page frames and so the system can avoid thrashing. Its pages are then often partially or full reclaimed as a result of the demands of the executing programs.

6 Concluding Remarks

The large memory systems of the next generation of computer systems pose a new set of challenges to operating systems memory management. While the traditional emphasis of virtual memory systems on statistically fair and efficient page replacement is becoming less important, the need has grown for direct control of physical memory by memory-bound applications as a result of the increasing speed of processors, the increasing size of physical memory and the increasing size of these applications and their data sets (and the lack of dramatic improvement in I/O throughput and response times). This paper has described ***external page-cache management as*** a way to address these issues.

External page-cache management provides an application view of physical memory as one or more physical page caches, allowing each application to create and explicitly manage the physical memory it has been allocated in an application-specific way. This abstraction contrasts with the conventional approach which' provides an abstraction of virtual address spaces, making the physical page cache transparent to the applications, except for its performance impact. Hiding actions and information from applications that make a difference in cost of hundreds of thousands of instruction times for a single memory reference is not reasonable and, by our design, not necessary.

External page-cache management has been implemented in the V++ system with relatively modest additions to the kernel. The changes lead to a significantly simplified kernel, because page reclamation, most copy-on-write support and distributed consistency can all be removed to the system page-cache manager. 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 version of the Unix **madvise** module which could deal with the memory management problems raised in this paper.

External page-cache management llows programs to successfully adapt to changes in the amount of physical memory available to them, leading to more efficient application and system execution. 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 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.

Our approach develops further a principle of operating system design we ***call efficient completeness,*** described previously in the context of supporting emulation [7]. The operating system kernel, in providing an abstraction of hardware resources, must provide efficient and complete access to the functionality and performance of the hardware. In the context of emulation, this approach is clearly necessary to be able to efficiently emulate whatever service interfaces are defined by the emulated system. In the context of memory-demanding applications, this approach is necessary so that the operating system never "stands between" the application and the performance that the hardware resources can potentially deliver.

Physical RAM memory is effectively used as a cache for secondary storage in modern virtual memory systems, mapped as pages by hardware memory mapping facilities. Thus, 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 physical page allocation among these page caches. The simple but restrictive transparent virtual address space of conventional system design can be provided by a standard page-cache manager, for conventionally structured applications. This principle of efficient completeness generally leads to a relatively low-level service interface, thereby being in concert with the goals of minimalist kernel design for kernel management of resources such as memory.

In summary, external page-cache management is a promising candidate for structuring the next generation of kernel virtual memory systems, addressing both the growing complexity of current conventional virtual memory systems and the growing demands of applications. In the short term, it seems feasible to extend the mapped file facility of Unix to provide application control and potentially stem the growing complexity of the Unix kernel. We expect the most exciting page-cache management results will 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.

7 Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211. This paper has benefited from comments and criticisms by members of the Distributed Systems Group at Stanford.

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*
In Proceedings of the 13th ACM Symposium on Operating Systems Principles, Pacific Grove, California, October 1991.

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

[4] David R. Cheriton
UIO: A Uniform I/O System Interface for Distributed ***Systems***
ACM Transactions on Computer Systems, 5(1):12-46, February 1987.

[5] 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.

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

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

[8] 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.

[9] Jeffrey L. Eppinger .
PhD Thesis, Carnegie Mellon University, February 1989.
Also available as Technical Report CMU-CS-89-115.

[10] 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.

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

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

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