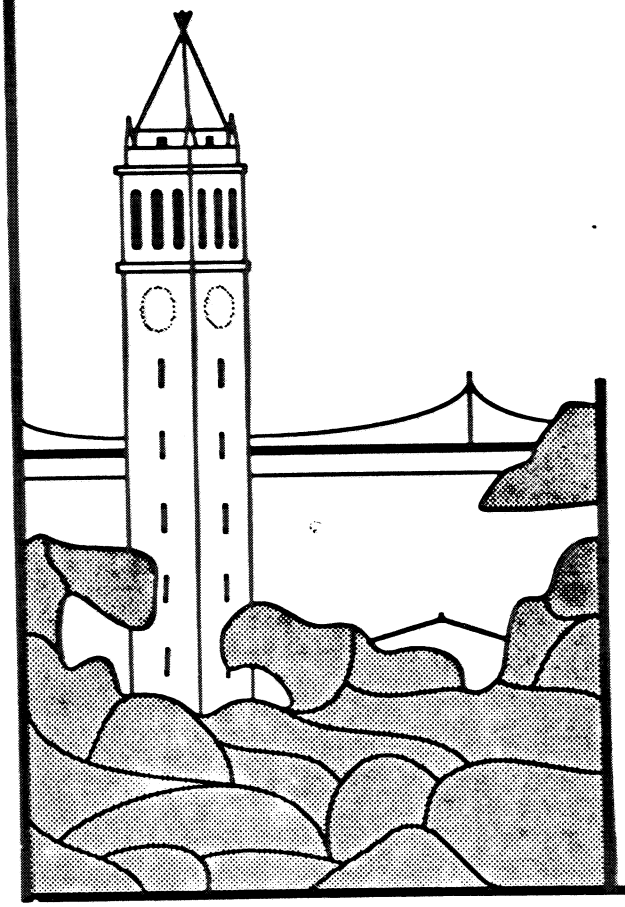
Tools and Strategies for the Development of Application-Specific Virtual Memory Management

Keith Alan Krueger



Report No. UCB/CSD-94-822

May 8th, 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Tools and Strategies for the Development of Application-Specific Virtual Memory Managers

Project paper for M.S.C.S.

Author: Keith Krueger

Advisor: Tom Anderson

Department of EECS, CS Division

University of California, Berkeley

Abstract

Although traditional virtual memory management policies perform well for most applications, several important applications, including database management systems, scientific applications, and multimedia systems, have poor paging performance. They access memory in unique ways that are not handled well by the default virtual memory manager. New approaches to the management of memory should be developed, that support *application-specific* tuning of the virtual memory management system.

Some existing systems address aspects of this problem, and allow application-specific memory management. They don't, however, provide adequate programmer support for the development of such management, and don't provide sufficient tool support for judging the memory access efficiency of applications.

To help software developers build application-specific virtual memory managers, the user-level virtual memory subsystem should include an easily extensible pager. Also, interactive graphical performance monitors should be provided to help programmers judge the effectiveness of their virtual memory management policy.

This paper describes current application-specific virtual memory systems and describes two tools that help in building application-specific virtual memory managers: (1) an extensible user-level pager, built on a virtual memory management system that supports application-specific tuning, and (2) a graphical performance analysis tool, called *VMProf*. These tools are demonstrated by using them to tune a scientific application.

1. Introduction

Operating systems attempt to efficiently utilize processor and storage resources, and at the same time allocate these resources fairly to competing applications. Operating systems hide the poor performance of input/output by using multiprogramming and virtual memory management. These general-purpose systems have been very successful in squeezing good performance out of computer systems for many years. A combination of new software requirements and recent developments in hardware technology, however, suggests that systems designers reconsider whether the general-purpose orientation of traditional operating systems is appropriate.

A perfect page replacement policy tosses out the in-memory page that is needed at the farthest point in the future (or one that's never needed again). Since it's not usually possible for a policy to predict every application's future use of pages, an imperfect policy that is highly accurate in predicting page usage is normally used instead.

The most common general-purpose page-replacement policy is an approximation to the *least-recently-used* (LRU) algorithm. The LRU policy manages a set of virtual memory pages by throwing out the in-memory page that hasn't been touched for the longest period of time. For applications that mostly fit into memory, LRU works well, because its policy decisions closely mirror the locality of the program's memory accesses. In most cases, it accesses the disk as few times as possible. This is increasingly important, because disk speeds are becoming slower relative to processor speeds, raising the cost of a page fault.

However, the LRU policy performs poorly for applications that access memory in non-standard ways, or those that exhibit locality on a scale larger than physical memory. Some of these headache-causing applications include:

- Scientific applications
- Database management systems
- Garbage collectors

- Multimedia systems

These applications all have memory requirements that can exceed the capacity of physical memory on a machine, but they don't exhibit the locality necessary for management with a general-purpose virtual memory page replacement policy (e.g. least-recently-used).

It is unlikely that changes in hardware systems will make it easier to manage memory for such applications. Adding more memory to a machine alleviates thrashing problems for applications that need slightly more physical memory than is available. However, many applications use widely varying amounts of memory. Using LRU with such programs often results in dramatic performance degradation when physical memory finally runs out; these applications would still benefit from a policy that degrades performance gradually as it runs out of physical memory. Since the cost of a page fault is rising, due to the relatively poor performance of disk storage systems, it is increasingly important to avoid unnecessary page faults. A policy that supports gradual performance degradation would avoid many faults. No obvious improvements in hardware technology will be able to offer the kind of improvement in performance that a good page replacement policy could.

One approach to building a improved page replacement policy is to attempt to build a "heroic" memory manager, that can handle a larger set of applications than LRU- (see [Hagmann 1992]). The difficulty in developing such a system is that the operating system page replacement policy must correctly anticipate the needs of all applications. Any change that benefits those programs that perform poorly under LRU may have an adverse effect on the performance of the vast majority of programs that do well with it, and thus hurt overall system performance.

A different approach allows each application to specify its own *application-specific* memory-management policy. Mach [Young et al 1987, Rashid et al 1988], V++ [Harty & Cheriton 1992], and Apertos [Yokote 1992] are all systems that implement this functionality. This approach offers the potential for near-optimal performance, but it requires more effort on the part of application developers than a general-purpose page-replacement policy. The operating system is responsible for allocating physical page frames among competing jobs; user-level pagers associated with each program decide which of the program's virtual pages are to be cached in the available physical memory. Any program which performs well with LRU can use the system's default policy; other programs can use a policy or policies tuned to their specific memory-access patterns.

Obviously, this approach gives the programmer more than enough rope to hang himself. The application-specific approach needs language facilities and user-interface tools to hide some of the difficulty of building application-specific page replacement modules. Software engineers should be able to diagnose problems with page replacement policies, easily change the policy, and measure the difference in performance. Notationally separating concerns about memory management from other program declarations assists programmers in evaluating their programs.

This paper describes a toolkit designed to make it easier for programmers to develop new application-specific page replacement policies. First, an extensible user-level virtual memory system is described. This system is object-oriented,

with disciplined entry points for non-expert programmers to easily modify key policy decisions. Second, a graphical analysis tool, *VMProf*, is described. VMProf allows a user to evaluate competing policies using a simple user-interface. These tools were used to tune the page replacement policy of several programs, using an instruction-level simulator to capture their paging behavior. A case study of successive over-relaxation (SOR), a scientific program, is used to illustrate the application of the virtual memory tools.

Section 2 describes why application-specific virtual memory management is useful. Section 3 describes the current support for user-level virtual memory management. Then, section 4 describes the extensible pager to be used for application-specific tuning. Section 5 describes the tool *VMProf*. Section 6 is a case study of how virtual memory problems related to successive over-relaxation may be alleviated using the virtual memory management toolkit. Section 7 describes related research. Section 8 is the conclusion.

2. Motivation

This section describes difficult situations for traditional virtual memory management systems. It also describes the difficulties and shortcomings of some of the suggested ways of improving performance.

2.1 Poorly performing applications

There are several general categories of applications that perform poorly with default LRU managers due to their non-standard memory reference patterns. These include scientific applications, database management systems, garbage collectors, and complex graphics systems.

It is common for all of these applications to solve memory-related performance problems by pinning a set of memory pages, and doing their own buffer management. This method works, but it does not integrate well in a multi-application environment, where there may be contention for those pinned pages.

2.1.1 Scientific applications

Scientists who use large data sets are accustomed to dealing with difficult memory management issues. To get accurate results, they often increase the size of matrices or other structures being used. However, many of these data sets do not fit into physical memory, and are usually not traversed in a way that can be managed effectively using an LRU policy. The data sets are often cycled several times from beginning to end. In such cases, LRU throws out the *least* recently used data page, even though it will be needed sooner than the *most* recently touched one (which will not be used again until a complete cycle through the data set).

It is common for scientific programmers to avoid virtual memory management altogether, opting instead to do memory management by hand. In fact, Cray supercomputers don't include any virtual memory management facilities. Many books have been written on how to properly code numerical problems; memory management is one of their chief concerns [Press 1989].

2.1.2 DB Management Systems

Database management systems also frequently deal with very large data sets, and have difficult memory management problems [Kearns & Defazio, 1989]. It is generally acceptable to manage the code of the DBMS using a standard LRU policy, but management of the data memory

is a different story. DBMS's often scan through very large data sets in a sequential or oddly-patterned way; the ideal memory management policy changes from operation to operation.

Modern DBMS's are often made up of varying computer memories and disks distributed across a network. A DBMS made up of these systems will not be efficient, unless decisions are made based on the costs associated with different memories (e.g. DRAM, flash RAM, battery-backed RAM) and disk access costs. DBMS's will only perform well in such environments if they have an application-specific virtual memory policy that offers accurate cost models.

The DBMS often understands how it will need to manage memory, but the operating system typically does not provide programmer-friendly facilities that allow the DBMS to inform it of its memory needs and get information about current memory allocation. As a workaround, database management systems allocate and control their own buffer spaces in physical memory, to avoid some of the problems associated with the general-purpose memory management policy of the operating system.

There are several patterns common to relational database access that are understood by the DBMS, but which are not managed well by an LRU policy. [Stonebraker 1981] suggests that operating systems should provide better support to database management systems. He says that the DBMS should simply give "advice" to the OS's virtual memory manager, relieving the DBMS of most of its buffer management responsibilities, and avoiding buffer management conflicts between the OS and the DBMS. An application-specific virtual memory manager would make this kind of tuning possible, and an extensible one would also allow the DBMS designer (or application designer) to easily add new memory management policies for different kinds of complex data structures that are not easily handled by relational access methods (e.g. spatial data, structured data types).

2.1.3 Garbage collection

Mark-and-sweep garbage collectors also access memory in a way that doesn't conform well to the least-recently-used page replacement policy [Alonzo & Appel 1990]. Once a page has been garbage-collected, it is not needed until the heap swings around again, yet LRU will keep it in memory because the page has been recently touched. If another program is running, and needs some pages, it could end up causing the swap-out of the next page to be heap-allocated, instead of swapping out the one needest the furthest into the future. Ideally, a memory manager for this application should give a high priority to the garbage collector's own code and data, so that regions of collected memory are always swapped out first. Further, the memory allocator should not re-use parts of the heap that have been swapped out until there is room for those pages to be swapped back in.

2.1.4 Graphics applications

Interactive graphics programs currently being developed require sophisticated memory-management techniques [Teller & Sequin 1991]. They often precompute vast amounts of information to enable real-time interaction. Access to this information is often sequential and the size of the needed information is frequently larger than available physical memory; this causes thrashing to occur under an LRU page replacement policy. This type

of application could produce more detailed images and exhibit much higher throughput if the page replacement policy of the operating system could be modified to fit the access-patterns of the program itself (e.g. through pre-fetching or an application-specific page replacement policy).

Multimedia applications, including HDTV applications, also require special-purpose memory management. These applications must effectively manage many megabytes of information in real-time. A least-recently-used policy is not appropriate for this kind of memory management. Virtual memory management systems should be flexible enough to accommodate application-specific page replacement policies for these applications.

2.1.5 Discussion

Many applications, including some of those described above, may be able to modify their run-time behavior according to the availability and cost of system resources [Harty & Cheriton 1992, Cheriton, et al. 1991]. For example, certain simulations generate a final result by averaging the results of a number of runs. Fewer runs of the simulation may be used to get an equally accurate result, if a larger sample size is used. If the simulator knows how much memory is available, it will be able to choose the sample size that fits into memory. Adapting application behavior to the amount of available physical memory is straightforward if there is user-level control over virtual memory, but it is very difficult or impossible if paging is implemented in the kernel, hidden from the application.

2.2 Discussion of proposals

Several alternative strategies to user-level virtual memory management have been proposed. These solutions are difficult to implement, and would be more widely used if they were made more palatable to the average programmer.

The idea of avoiding or fighting the operating system's built-in memory management is common to most of the solutions. The virtual memory client's (programmer's) point of view is not addressed very well by any of them; software engineers have to buy more memory or think like an operating system designer to use any of them.

2.2.1 Buy more memory

Many times, thrashing may be avoided most simply by buying more memory. This is attractive because memory is cheap and getting cheaper. However, many application programmers typically would like to increase the size of the data set that they use, if possible. Irrespective of the amount of physical memory, many problems will require more memory than the physical memory can support [Hagmann 1992].

Some users push the performance of their computer systems until the machine starts to thrash. In such situations, when physical memory needs are nearly met, a carefully designed memory management policy could prevent thrashing by allowing the system to degrade more gracefully than LRU.

When data sets are several orders of magnitude larger than the size of any affordable physical memory, neither buying more memory, nor an application-specific policy will prevent thrashing. However, using a performance analysis tool and extensible virtual memory system

may be able to marginally improve the performance or assist in the re-design of the application's data access methods. (Section 6 demonstrates how to handle this situation).

2.2.2 Restructure the application to improve locality of reference

Often, programmers resort to completely restructuring their code to improve spatial and temporal locality. This technique, known as "blocking", is commonly used in scientific code to improve processor cache performance. It may also be used similarly to improve virtual memory performance. Effective blocking often requires making the program structure more complex. This complexity often obscures the problem being solved, and it is very difficult to block code without making it difficult for humans to read. Applications with very clear memory-access patterns, such as matrix scans, are relatively easy to block. Applications with irregular or frequently-changing access patterns are very difficult (although not impossible) for programmers to block.

VMProf, the virtual memory profiler described in section 5, could help programmers quickly come up with good blocking code, by providing a visual representation of the data being managed by the blocked code. However, VMProf doesn't reduce the complexity of blocked code. A well-structured, extensible user-level pager, on the other hand, could help separate some of the memory management concerns from the main code, reducing (if not eliminating) the blocking needed for the code without reducing its performance. This partial separation of memory management concerns from other program concerns could make the blocked code less difficult to write. Part of the case study in section 6 shows how blocking can be combined with virtual memory tools such as VMProf and an extensible user-level pager to get both improved performance and improved human-readability.

2.2.3 Pin and manage a pool of the operating system's memory

[Stonebraker 1981] suggests bypassing the operating system's virtual memory system by pinning a pool of the application's pages into physical memory. This solution is frequently used by DBMS's. The user code explicitly manages the buffer pool as a cache for disk by deciding which disk pages get swapped into main memory.

This solution works fairly well, but there are some problems with it. Developers must spend a lot of time building this kind of manager; user-friendly tools could reduce this time. Also, all accesses to program data must go indirectly through the user-written buffer manager. Lastly, it is inflexible in a multiprogrammed environment [Harty & Cheriton 1993].

Good tools for virtual memory management should separate memory management concerns from other program code. Also, they should allow for code re-use and a quick analysis of memory usage. It should be possible to notationally separate concerns about memory management from the program itself. The tools described in the sections that follow allow a programmer to write their main code in terms of normal memory reads and writes, and to write or extend a *separate* virtual memory management module.

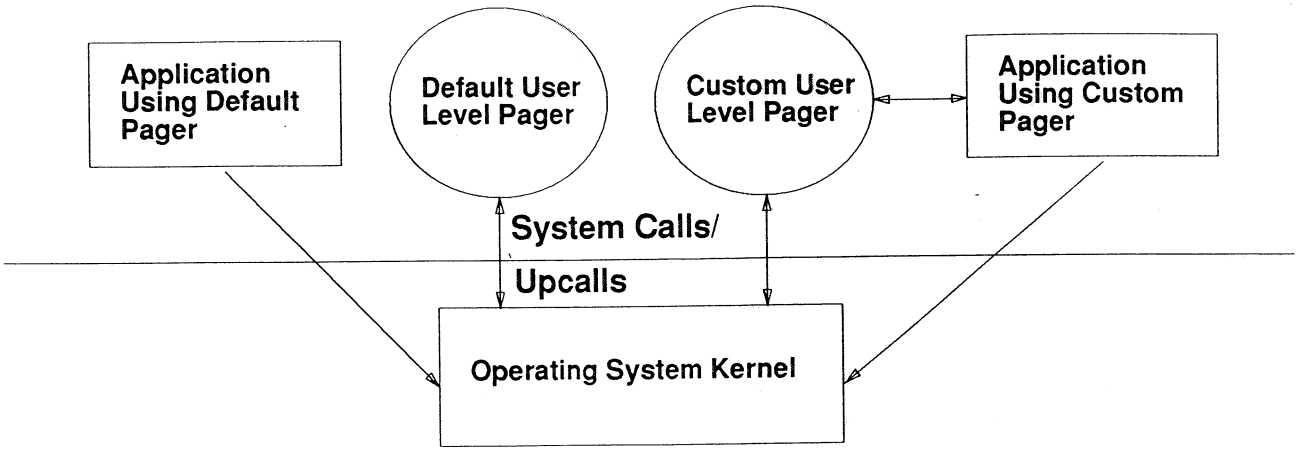


Figure 1: The interactions of applications, their pagers, and the kernel

3. Kernel Support for User-Level Virtual Memory Management

The traditional kernel-encapsulated virtual-memory management system sometimes forces application developers to devise elaborate ways of accessing memory, to adjust to the expectations of the default LRU page replacement policy.

An *application-specific* memory management systems moves virtual memory management policy from the kernel to the user-level, and allows programmers to separate the issue of how to *use* the virtual memory management abstraction from the issue of how to *implement* it. This separation allows those clients who do not need to adjust their policy decisions to continue to use the simple default abstraction. It also means that even in those cases where programmers must work around the default policies (e.g. databases, garbage collectors, scientific applications), they can cleanly divide their attention between working with the application, and working with the basic virtual memory abstraction.

Application-specific virtual memory management is a departure from traditional virtual memory management and requires different kernel support. There are several systems that provide kernel support for user-level virtual memory management. These include Mach, extended by [McNamee & Armstrong 1990], V++, and Apertos. These kernel implementations have many features in common, which are described below in terms of the responsibilities of and the interactions between the kernel, the application, and the application-specific memory manager.

3.1 Division of responsibilities

Responsibility for virtual memory management is divided between the kernel, the application, and a separate, application-specific user-level pager. This division of responsibility is displayed in Figure 1. In the simplest (and most common) case, an application needs no special-purpose paging policy, and can use one of the default pager policies; no changes to the application code are needed and no extension of the paging policy is needed. In more complicated cases, it may be necessary to change the paging policy or use more than one policy to get good performance. In the most difficult cases, it may be necessary to modify or re-write one or more paging

policies and also make changes to the program body to get efficient use of resources.

The operating system kernel allows the control of paging policy to be associated with individual applications. Unlike a traditional, monolithic organization, the kernel is responsible only for allocating physical pages among competing jobs and for providing a mechanism for user-level pagers to modify page tables. Each user-level manager is given a set of physical pages to manage by the kernel, and it decides which of the application's virtual pages are assigned to physical pages and which are assigned to disk. In this way, the *mechanisms* necessary to implement virtual memory are separated from the application-specific *policy* implemented at the user level; the kernel is responsible only for providing a set of building blocks and tools that may be used to make policy decisions.

In a traditional OS kernel, a kernel routine is invoked when a page replacement decision is needed. To support user-level VM management, this decision is no longer done in the kernel; instead, an *upcall* is made to the user-level pager associated with the application of the faulting page. For instance, on a page fault, the kernel upcalls to the user-level pager, providing any needed information (such as hardware page reference information) to the user-level pager. The user-level pager then chooses which page to replace. In addition to making upcalls on a page fault, the kernel must inform the user-level pager of any changes in resources that could affect the way memory is managed at the user level (e.g. the number of physical memory pages assigned to an application could change) [Anderson et al 1992].

In addition, a sophisticated application may have a communication channel to the user-level pager. The application can inform the virtual memory system in advance of phase changes where a different policy might be used; the virtual memory system can inform the application of any increase or decrease in the amount of available physical memory, to allow the application to adapt its behavior appropriately.

3.2 Interactions between applications, user-level pagers, and kernel

The interactions between the application, the operating system's kernel, and the user-level pager are represented in Figure 1 and are described as follows:

- When a page fault occurs for a virtual memory access, it triggers a trap into the operating system. The kernel makes an up-call to the application's pager with the faulting virtual address. This upcall then dispatches the user-level method **HandlePageFault** within the user-level pager. This procedure is responsible for placing the disk-bound page into physical memory (through system calls back to the operating system).
- If all the process's physical pages are allocated, the pager is also responsible for choosing the in-memory page to swap out. The user-level pager polls the operating system to obtain information (e.g. hardware page usage and modified bits) regarding a process's page access patterns. This information is then used by the user level paging policy when choosing the page to replace.
- Modified in-memory pages that

are chosen for replacement must be written back to disk. System calls to the OS accomplish this.

- If an application wishes to change its paging policy at run-time, or if it needs to know which of its pages are mapped to physical memory, it makes requests over a communication channel between the user-level pager and the user application.
- When the resources (e.g. memory) change, the kernel notifies the user-level pager using an upcall. If the user-level pager needs more or fewer physical pages, it makes a system call to the kernel [Harty & Cheriton 1993].
- The user-level pager can make a special request that the kernel unmap a page but not remove it from memory, causing the application to trap on read or write references to selected pages. This capability can be used to collect more detailed page usage information [Levy & Lipman 1982] or to provide other features such as distributed virtual memory, transactional memory, or automatic checkpointing [Appel & Li 1991].

It is important to keep in mind that there is not a one-to-one mapping between applications and paging policies. A single default user-level pager may be used by all of the applications that perform well with an LRU policy. At the other end of the spectrum, a single application may use a variety of paging policies for different segments of memory or for different phases of the program's execution.

Ideally, the operating system should include a set of the most popular paging policies, allow for "plug-and-play" use of them, and also allow for easy extensions to be made to them. Application-specific virtual memory will only be used if it is easy for the programmer to understand.

4. An extensible user-level pager

Although many systems already provide user-level virtual memory management, they do not include enough user support to encourage their use. There are many applications that would benefit from application-specific virtual memory management. Plenty of programmers would like to avoid mixing memory management issues into their code. The user-friendly operating system structures and tools described in the paper's remaining sections make it easier to build and analyze application-specific virtual memory managers. This section describes the first of the two user tools for managing virtual memory: the extensible user-level pager.

The extensible user-level pager hides as much complexity from the programmer as possible, while allowing the most performance-critical routines to be modified for each application's peculiar access patterns. There are two objectives for the pager. First, it should be built on a useful set of built-in virtual memory management facilities that perform tasks common to all memory managers; a large amount of the code for traditional virtual memory systems is taken up with bookkeeping code and other common code. Second, the structure of the pager should expose the most commonly-needed interfaces and values to the user, so that they may be easily changed or extended. This was done by structuring the code into classes and methods, and allowing programmers to selectively inherit or rewrite code based on the default classes. This allows the construction of a new implementation containing only the essential changes, following the protocol pre-

sented by the extensible user-level pager [Bershad et al 1988, Kiczales et al 1991].

The pager protocol described below focuses on the interfaces that help application programmers to develop efficient paging systems, and which hide the issues related to inter-application contention for resources. As far as an individual application's pager is concerned, it has a segment of physical memory guaranteed to it; other, system-wide, policies decide how to divide physical memory among competing programs. Issues related to inter-application memory management are addressed in [Harty and Cheriton 1993], who suggest some cost models for physical memory distribution. This section shows how to help the application designer's manage a single application's physical memory/virtual memory mapping problem.

4.1 Description of the extensible user-level pager

The routines and classes that follow are the defaults provided by the operating system. To improve the performance of an application, a programmer simply rewrites the exact parts of the system that aren't appropriate for the application. There is no need to re-implement the entire pager, or to learn hundreds of lines of code. The set of interfaces making up the virtual memory pager protocol is sufficiently simple that only one or two routines should have to change to improve the performance of an application.

The user-level pager consists of a group of extensible classes and their methods. This section describes how the paging system works, in general. Section 4.2 gives more details on the classes and interfaces that make up the extensible user level pager.

Memory management is tuned on a per-process basis; the virtual memory management protocol is structured as follows:

- Each process has a **ResidentPageTable** associated with it which contains information about the *physical* pages assigned to the process by the kernel (e.g. the identity of the virtual page contained in each physical page). Typically, there are three lists of physical pages: a list of unallocated pages, a list of deallocated but mapped pages (providing a "second-chance cache [Levy & Lipman 1982]), and a list of the allocated and mapped pages. The kernel has the authority to decide how many physical pages to assign to a page table, while the user-level process controls how to map virtual pages onto them.
- Each process gets an **AddressMap** object, which encapsulates the information about the *virtual* pages of the process. It is similar to a traditional page table. It contains a virtual-to-physical map, and hardware reference information (e.g. dirty bits).
- When a page fault is triggered, the **HandlePageFault** method of the **AddressMap** class is messaged. This message originates in the kernel and travels via an upcall to the user level.
- If there are no available physical pages, the **HandlePageFault** method calls on **FindPageToReplace** to pick a victim page to unmap from memory.
- The user level pager may query the kernel for information using the method **PollKernel**. This allows

user-level pagers and performance tuners to obtain better metrics when making policy decisions.

4.2 Programmer's view of the virtual memory management system

Section 4.1 described how the kernel and user-level interactions occur. If a programmer keeps this knowledge in mind, it is easy to extend the user-level pager. This section describes the C++ interfaces that programmers have to understand in order to derive a new application-specific pager from the default ones provided by the operating system.

The classes and methods described below are simple, but provide enough flexibility that significant performance improvements may be made by extending them. The routines include those for the resident page table:

```
class RPTE // {"ResidentPageTableEntry"
        int physicalFrame;
        AddressMapEntry* virtualPage;
        Bool free;
};

class ResidentPageTable {
public: // Upcalls from the kernel
        void AddPageToAllocation
                (int pageNum);
        void RemovePageFromAllocation
                (int pageNum);
        int FindFreePage();
        int GetNumFreePages();
private:
        int tableSize;
        RPTEList* allocatedPages;
        RPTEList* unmappedPages;
        RPTEList* freePages;
};
```

When the kernel gives out physical pages, it associates one **RPTE** (resident page table entry) with each of them. The integer **physicalFrame** is the physical page number of the page; it is metadata describing the virtual memory system which is used by both the pager and the kernel. The **ResidentPageTable** class lists the physical pages allotted to a process. This list may change at run time; if this happens, the kernel uses the methods (upcalls) **RemovePageFromAllocation** and **AddPageToAllocation** to notify the user-level pager of the change in the number of physical pages it may manage.

The **AddressMap** class encapsulates both a process's virtual memory and state information for its page replacement policy. Each page of virtual memory has an **AddressMapEntry**. The classes are defined as follows:

```
class AddressMapEntry {
public:
        RPTE* physicalFrame;
        Bool valid;
        Bool modified;
        Bool used;
};
class AddressMap {
public:
        virtual int FindPageToReplace();
        virtual void HandlePageFault
                (int faultPage);
        virtual void FetchPage
                (int targetPage,
                 int faultPage);
        virtual void PollKernel();
private:
        AddressMapEntry* pageTable;
        int pageTableSize;
        ResidentPageTable coremap;
        int LRUClockHand; // for LRU policy
};
```

Typically, the user-level pager periodically calls **PollKernel** to determine the process's recent page access patterns.

PageIn reads the **faultPage** from the backing store and stores it in **pageToReplace**, writing the **pageToReplace** to the backing store if modified.

In the simplest case, the user-level virtual memory manager consists of an **AddressMap** instance and a **ResidentPageTable** instance. The programmer creates a new paging policy by changing the methods for these objects, compiling a new memory manager and asking the kernel to associate the new manager with the application. More complicated paging systems made up of multiple policies can also be coded into the same manager, allowing applications to change their page replacement policy as their memory access patterns change; for applications that can predict their future memory accesses, this capability is very effective in reducing the number of page faults.

The following methods are the standard ones provided with the memory managers to implement the default approximation to LRU. They may be extended or replaced with more effective methods, depending on application performance requirements.

```
void
AddressMap::HandlePageFault
        (int faultPage)
{
 int pageToReplace;
 if (coremap.GetNumFreePages()==0)
        pageToSwap= FindPageToSwapOut();
 else
        pageToSwap= coremap.FindFreePage();
 PageIn(pageToSwap, faultPage);
};
```

```
// Implementation of a one-bit clock
// algorithm approximating LRU
int
AddressMap::FindPageToReplace()
{
 while (1) // loop until page is found
 {
        LRUClockHand++;
        if (pageTable[LRUClockHand].valid)
         if (!pageTable[LRUClockHand].used)
                return LRUClockHand;
         else
                pageTable[LRUClocHand].used
                        = FALSE;
        if (LRUClockHand == pageTableSize)
                LRUClockHand = 0;
 }
};
```

4.3 Extending the user-level pager

To improve the performance of an application, a programmer modifies those parts of the system that don't perform well for the application. The set of interfaces provided by the user-level pager is sufficiently simple that only one or two routines should have to change. For example, it could be that a most-recently-used (MRU) page replacement policy is appropriate for managing some memory. (This would be the case for a cyclic sequential scan of a large amount of memory, e.g. a database *join*). Parts of the paging system could be changed by inheriting the default pager class, and then modifying the parts that need changes to improve performance. To derive an MRU policy using the user-level pager interfaces, the following simple code changes would suffice:

```
class MRUAddressMap : public AddressMap {
public:
        int findPageToReplace();
private:
        int MRUClockHand;
};

// One-bit approximation of MRU
int
MRUAddressMap::findPageToReplace()
{
 while (1) // loop until page is found
 {
        MRUClockHand++;
        if (pageTable[MRUClockHand].valid)
         if (!pageTable[MRUClockHand].used)
         {
                for ( int i=0;
                        i<pageTableSize; i++)
                 pageTable[i].used = FALSE;
                return MRUClockHand;
         };
        if (MRUClockHand == pageTableSize)
                MRUClockHand = 0;
 };
};
```

The new method implementation for **findPageToReplace** is now called when a page fault occurs in a process running in an **MRUAddressSpace**. To implement an MRU policy through the protocol, the programmer simply creates one new subclass and modifies one documented function. The details of other functions and classes were unchanged. Similarly, **HandlePageFault** could be modified to pin and pre-fetch memory pages.

The programmer may also wish to provide multiple page replacement policies for an application. For example, a DBMS may wish to use an LRU policy on the code segment, and use several different access methods (i.e. page replacement policies) on the data being scanned. In this case, the programmer simply creates a sub-class of the **ResidentPageTable** class which breaks up the allocated physical pages into lists corresponding to the different logical segments. The **HandlePageFault** routine is then changed to call an appropriate derivative of **findPageToReplace**, according to the identity of the page that faulted. In this way, the user-level memory manager is better integrated with the DBMS.

5. VMprof -- The Virtual Memory Profiling Tool

Given the complex trade-offs involved with the virtual memory system, it is not enough to simply give the user control over the implementation. Tools must also be provided to help identify performance problems with the particular application/policy combination, to help identify ways to improve performance. Also, the user needs to be able to quickly evaluate the performance effect of changes to the application or the paging policy.

VMprof, the virtual memory profiling tool, allows the programmer to easily identify problems with virtual memory performance by displaying the dynamic behavior of the paging system graphically with an interactive tool. The user-level extensible pager and VMprof complement each other by decreasing the time needed to tune virtual memory system performance.

VMprof supplements other program performance analysis tools such as UNIX gprof [Graham et al 1982] and MemSpy [Martonosi et al 1992]. A trace of page faults is generated (using a modified version of *qpt*, an instruction-level simulator [Larus 1993]) and fed into VMProf. VMprof allows the user to analyze both spatial and temporal aspects of the virtual memory management. VMprof's

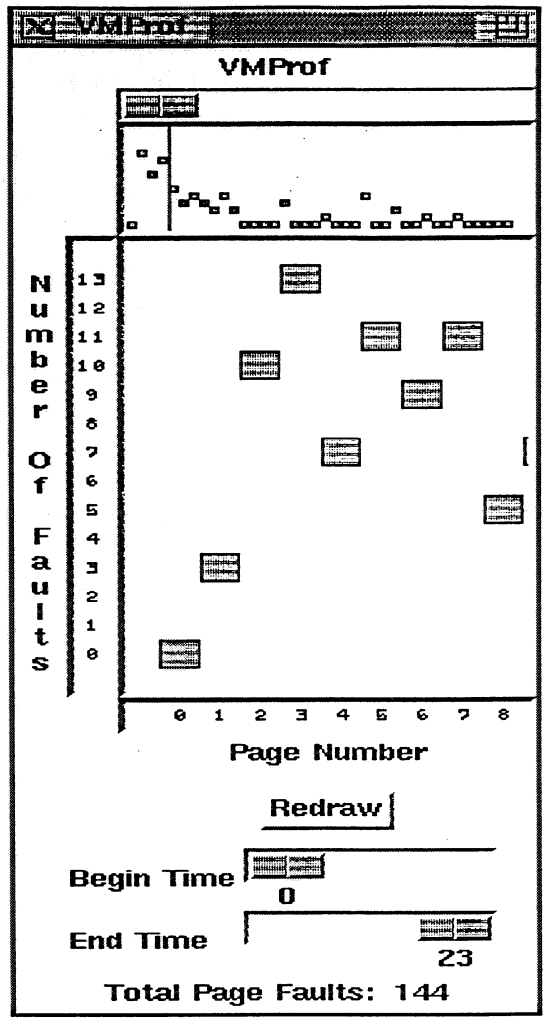


Figure 2: Screen display of *VMProf*, the virtual memory profiler

graphs may be used to identify regions of the address space with high page fault rates. By adjusting the time frame, a user may also investigate how fault behavior develops with respect to time. The graphical nature of VMprof facilitates quick analysis and improvement of virtual memory performance.

Figure 2 shows the output of the VMprof virtual memory profiler. The top graph is a histogram of page faults in virtual memory. The horizontal dimension reflects sections of virtual memory, from low memory on the left to high memory on the right. The vertical dimension represents the frequency of page faults for each section of virtual memory. Because there can be a large number of virtual pages under consideration, each point on the graph refers to the aggregate number of faults for a contiguous range of pages. The bottom graph displays fault behavior at a (configurable) finer level of detail than the global view of the top graph. If a user notices that there is an interesting

pattern in the global display, the scroll bar may be moved to focus the local display on the desired region.

Behavior with respect to time may be displayed by moving a pair of sliders: **begin-time** and **end-time**. Only the page faults occurring in this time frame are displayed in the two graphs. Programmers use this feature to isolate portions of the program and judge whether they would benefit from modifications to the paging policy. The time sliders may be used to move slowly through time to see how page-fault patterns develop. The spatial and temporal aspects of memory access patterns may be evaluated by adjusting the local view of page fault behavior and the time frame under consideration.

Experience using VMprof suggests improvements that would make it more useful. One would be to more closely connect the application's symbolic program constructs and the output of the virtual memory profiler, creating a combination debugger/performance analyzer. Currently, VMprof's display offers enough information for a rough view of memory access patterns. A more useful tool would allow the user to select the particular memory objects to observe and to place "breakpoints" in the program code that would separate segments of the code that exhibit different memory access patterns.

In addition, the user should be able to easily select memory objects to observe, using their symbolic names or icons, and associate a virtual memory policy with each one. Also, it should be possible to use a separate tool to see how multiple programs interact when sharing the same physical memory resources, for those programs that adjust their memory usage based on run-time conditions. For instance, [Harty & Cheriton 1993] suggest that programs "bid" for physical memory; the kernel can then use a market approach to system memory allocation. Using VMprof, experiments could be performed interactively with the profiler to see how different virtual memory policies perform in isolation and in tandem, with different memory allocation arbitration.

6. A case study: Using VM tools with successive over-relaxation

This section illustrates how to use the extensible user-level pager and VMProf to improve a poorly-performing application. It describes the analysis and improvement of successive over-relaxation (SOR), an operation common to many scientific applications. This example is fairly simple, for illustrative purposes, but the same extensible pager and tool could be used with more complex examples as well. The performance of SOR is greatly improved through a combined use of VMProf and the extensible virtual memory pager.

To analyze SOR's performance, memory reference traces were gathered for a basic implementation of the SOR application using several different problem sizes. The application's paging behavior was determined by feeding these traces through a simulator which invoked the user-level policy module on each page fault. The resulting page fault sequence was fed into VMprof; this made it possible to quickly identify problems with the application's virtual memory performance.

In successive over-relaxation, each element of a matrix is averaged with its four immediate neighbors (called a "relaxation step"). This operation is repeated on the matrix until a steady state for the matrix's values is reached. The following code fragment is a simplified (ignoring boundary conditions) version of SOR:

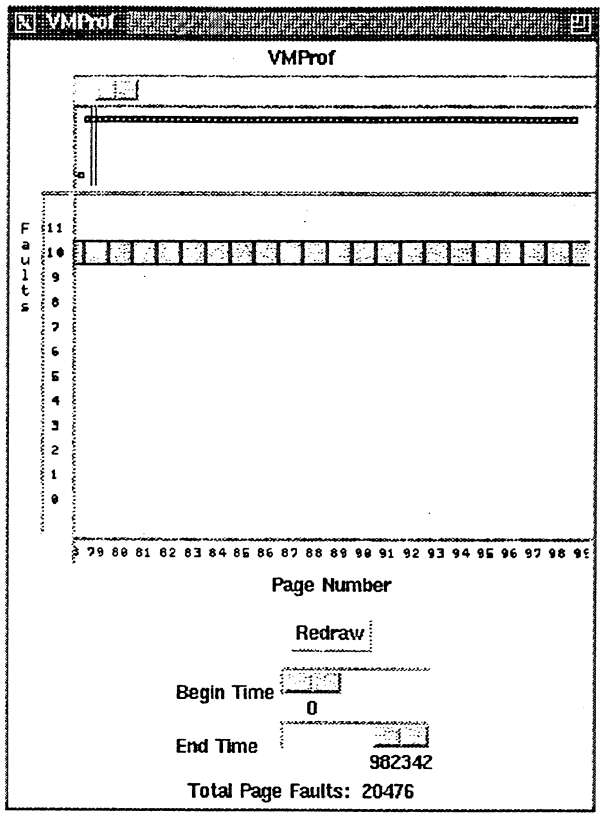


Figure 3: SOR with an LRU page replacement policy

```
while (!converged) // Outer loop
for (r = 0; r < matrixRows; r++)
 for (c = 0; c < matrixCols; c++)
       matrix[r][c]=(matrix[r-1][c] +
                     matrix[r][c+1] +
                     matrix[r+1][c] +
                     matrix[r][c-1]) / 4;
```

Figure 3 displays the output of the VMprof tool, profiling the initial SOR implementation with an LRU page replacement policy. The matrix size was chosen to be 1K by 1K; each element is a double precision floating point number. In all, the matrix occupied 8 megabytes of virtual memory. It was assumed, for this example, that there were 8 megabytes (2048 4KByte pages) of physical memory available. Since the application code also takes a small amount of space, the program and its data together didn't fit in physical memory.

Figure 3 shows how the cyclic sequential access pattern of SOR, combined with LRU, results in a large number of faults (this pattern is also common in DBMS applications). The row labeled **faults** shows that page faults are frequent: there is one fault per iteration of the loop per page of data whenever the size of virtual memory is larger than the amount of physical memory. A user watching the number of page faults updated with respect to time sees a continuous left-to-right cascading of faults, which suggests that thrashing is occurring.

The performance of LRU is much worse than optimal in this situation. VMProf graphically shows the access pattern for

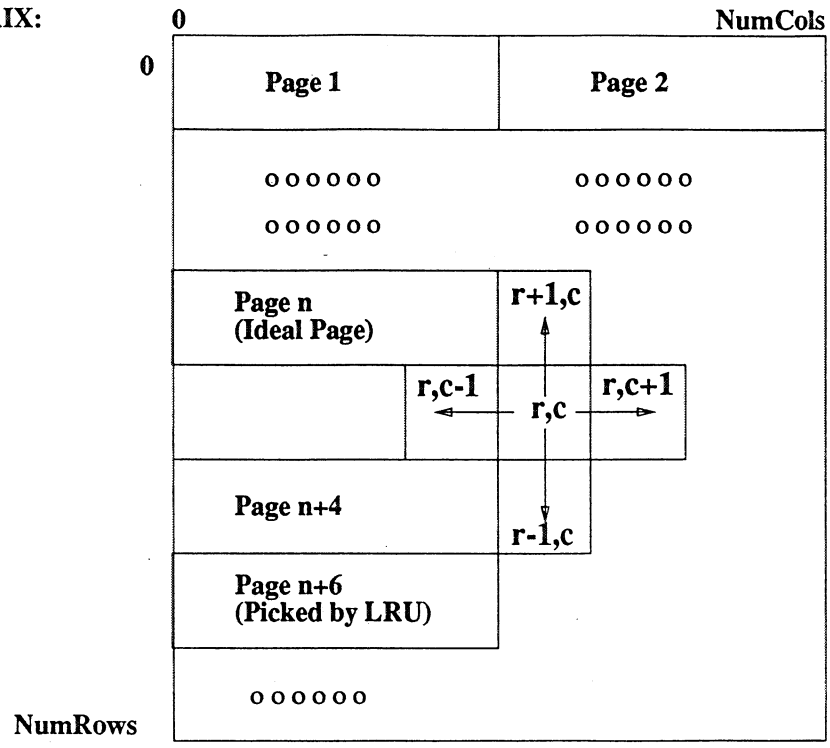


Figure 4: The matrix layout for successive over-relaxation

SOR in Figure 4. During the first iteration of the outer loop, there are 2K compulsory page faults, since none of the pages have been previously accessed; these page faults would occur and cost the same with any page replacement policy, unless pre-fetching techniques were employed to avoid some disk seek costs.

For iterations following the compulsive page faults of the outer loop, however, the number of page faults depends greatly on the virtual memory policy used. If calculating the value for **matrix[r][c]** causes a page fault in reading **matrix[r+1][c]**, the LRU policy will choose the physical page which has not been used for the longest period of time (page **n+6** in Figure 4, assuming each row of the matrix takes 2 pages of memory). Unfortunately, due to the cyclic sequential memory access pattern, this will be the very next page accessed in the matrix, and this access will once again cause a page fault. As indicated by VMprof, LRU results in a page fault on every page of the matrix for each iteration through the outer loop.

An ideal page replacement policy replaces the page which will not be needed for the longest time in the future. When reading **matrix[r+1][c]** causes a page fault, the page which will not be referenced for the longest time in the future is the first full page located immediately before **matrix[r-1][c]**. It is clear that a custom page handler should choose virtual page **n** from Figure 4 for replacement.

A custom page replacement policy that is specialized to the access pattern of SOR may be quickly created by simply writing a new version of **FindPageToReplace**. This custom policy has to be tailored to the size of the array and the machine's page size. In this example, if a page fault occurs in accessing virtual page **v**, then the ideal page to replace is the physical

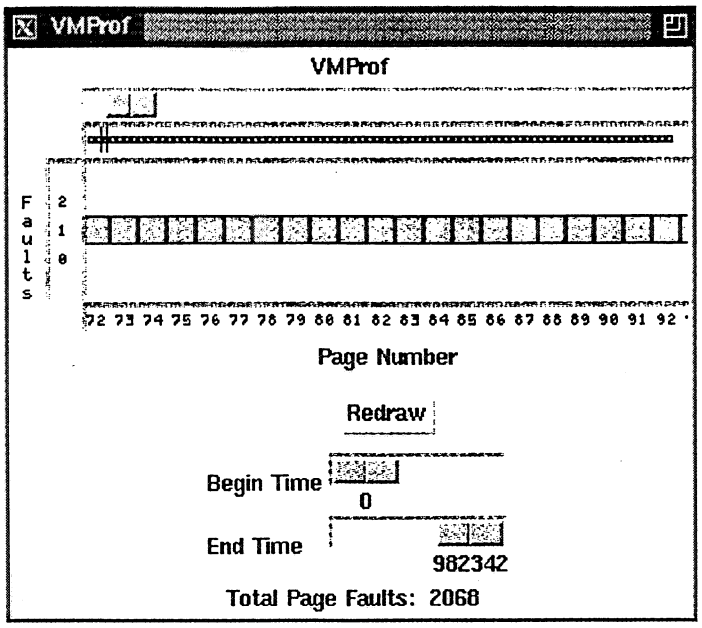


Figure 5: SOR with a customized page replacement policy

page containing virtual page **v-5** (since there are 2 pages per row). The custom policy has the flavor of the most-recently-used (MRU) page replacement policy; it treats the code segment with an LRU policy while treating the data segment with an MRU policy.

This custom replacement policy is implemented by slightly modifying the implementation of MRU. The address trace of the SOR is run again through the simulator, invoking the *custom* paging policy. The resulting VMprof output is displayed in Figure 5. Using this policy, after the compulsive start-up costs of faulting the array into main memory, there is only one fault per iteration of the outer loop (as opposed to one fault per page per iteration). The result is a large reduction in the number of page faults.

The magnitude of advantage of the custom paging policy relative to LRU depends on the difference between the virtual memory needed and the available physical memory. Using LRU, the application thrashes *immediately* when it needs more virtual memory than the available physical memory. Using the custom, MRU-flavored custom policy, performance degrades more gracefully.

Eventually, of course, if the matrix size is much larger than will fit in virtual memory, even the custom policy will perform slowly, although optimally. Figure 6 shows the number of page faults incurred by SOR (z-axis) as a function of the number of iterations of the outer loop (x-axis) and the difference between available physical memory and required virtual memory (y-axis). The plot shows that the number of page faults for the custom policy is dependent on the relative amount of physical and virtual memory. The performance of the LRU policy is uniformly poor, independent of the number of available physical pages. The plot also shows that as the number of available physical pages decreases, the customized policy's

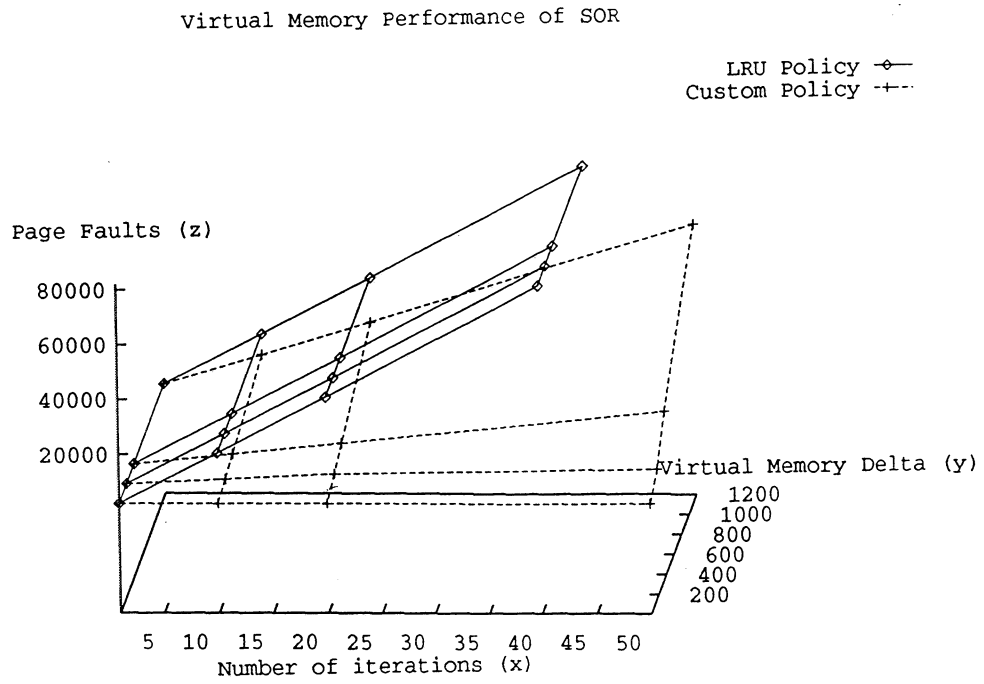


Figure 6: Number of page faults of SOR with different policies

performance begins to approach that of the least-recently-used policy.

Modifying the paging policy by itself does not help when the matrix is very large with respect to the amount of physical memory. Instead, to get good performance in such a situation, the application's implementation should also be changed to exhibit more spatial and temporal locality. The first implementation scanned the entire matrix from beginning to end for each and every relaxation step. For example, if it took 10 relaxation steps for the matrix to converge, each page of memory would be crossed 10 times using the original code.

Instead, to improve performance, a "blocked" implementation of SOR could be used, in which several relaxation steps are performed during a single sweep through the array. For instance, after computing the relaxation for rows **1** to **r** during iteration **i**, the implementation can immediately compute the next iteration for rows **1** to **r-1**, without changing the original data dependency ordering of the application. As long as **r** is smaller than the size of physical memory, the program can compute more relaxation steps for the same number of page faults relative to the original implementation.

Even the blocked version of SOR benefits from a custom page replacement policy in some cases. The blocked implementation must still scan multiple times through the entire array to complete the relaxation. As with the original version of SOR, if the size of the matrix is slightly bigger than the amount of physical memory, LRU will tend to throw out pages that are about to be scanned, while a custom policy can be easily devised to throw out pages that are not needed for the longest time into the future.

Pre-fetching could further improve the performance of the SOR application. Prefetching is most useful when an application accesses a large number of pages in

sequence. Rather than having to fault each new page in turn, pages can be brought into physical memory before they are needed. The application would still be limited by disk bandwidth, but it would incur less overhead waiting for faults to be serviced.

In summary, the following steps are typically needed to tune a program to decrease the number of page faults:

- Identify and isolate phase transitions in the program using the visual cues provided by the VMprof performance tool.
- Experiment with different page-replacement policies by modifying the extensible user-level paging system. Use VMprof to determine the policy most appropriate for the observed access pattern for each phase of the program.
- If performance is still not good enough, write a "blocked" version of the program. Use VMprof to determine whether it performs well using LRU or still requires a custom policy.

The methods described above can be used in combination to switch policies on the fly by communication with the user-level pager during execution of a program. In this way, phase transitions that occur during execution may be matched with appropriate management policies.

VMprof may be used to determine memory access patterns of different phases and evaluate modifications to user-level virtual memory policy and algorithm implementation. Proper use of the well-defined programmer's interface to virtual memory and the simple VMprof profiler improves the speed with which efficient programs may be built.

7. Discussion

Some researchers interested in meta-level architectures suggest that the problems addressed here are not unique to virtual memory management systems. The focus of most open implementation work has been to develop frameworks through which those systems that need to expose aspects of their implementation can do so in a clean, modular and maintainable fashion.

Application-specific virtual memory management is an instance of a larger trend towards structuring system software to allow application control over policy decisions. Other examples include thread scheduling [Anderson 1992], interprocess communication [Bershad 1991], compiler optimizations [Steele Jr. 1990], database access routines [Dewitt & Carey 1984, Stonebraker 1987], and desktop publishing [Aldus 1992, Clark 1992, Dyson 1992]. In all of these cases, allowing the application to control policy offers the potential for more flexibility and better performance, in part because it is difficult to design a complex system to be optimal for all users of the system.

It is important to accompany these application-specific system software designs with well-designed, user-oriented performance monitors. A programmer cannot always rely on deriving new implementations from the provided system facilities when faced with inadequate performance. With virtual memory, there are times when the lack of available memory causes very poor application performance, irrespective of the paging policy (even the *optimal* page replacement policy for a given application results in poor performance). In these cases, it is necessary to rewrite the application to use a smaller working set. Tools such as VMprof can be used to identify such instances and to

suggest ways in which the application can be rewritten to improve performance.

Several research efforts in the operating system community have looked at making various pieces of the operating system customizable. Apertos [Yokote 1992] is designed to be entirely reflective to allow every part of the operating system to be under application control. Presto [Bershad et al. 1988] is a user-level thread system linked into parallel applications as a runtime library; because different applications might need different thread scheduling policies, Presto was structured to make scheduling easy for users to change. Work is also being done to define a good framework for multi-application (i.e. runtime-environment-specific) allocation of memory and other resources [Harty & Cheriton 1993].

8. Conclusion

The combination of the extensible, user-level page replacement module and the graphical analysis tool VMProf helps programmers to tune virtual memory performance. Together, they allow users to easily experiment with various page replacement policies and to get quick feedback from the user interface. This feedback may be used to develop a paging policy that better meets the application's demands, or, in some cases, to modify the application to exhibit better paging behavior. The programmer's interface of the user-level pager allows users to easily modify parts of a complex system, and the graphical user interface of VMProf provides a facility that eases the evaluation of the different trade-offs involved with modifying operating system policy.

9. References

[Aldus 1992] Aldus Corporation. Seattle, WA. *Pagemaker Additions Developer Toolkit*, February 1992.

[Alonso & Appel 1990] An Advisor for Flexible Working Sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 153-159, May 1990.

[Anderson et al. 1992] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pp. 53-79, February 1992.

[Appel & Li 1991] Appel, A. W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 96-107. Santa Clara, California, April 1991.

[Bershad et al. 1988] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A System for Object-Oriented Parallel Programming. *Software--Practice and Experience*, 18(8):713-732. August 1988.

[Bershad 1991] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175-198, May 1991.

[Cheriton et al. 1991] Cheriton, D. R., Goosen, H. A. and Machanick, P. Restructuring a Parallel Simulation to Improve Shared MEmory Multi-processor Cache Behavior: A First Experience. In *Shared Memory Multiprocessor Symposium*, Tokyo, Japan, April 1991.

[Clark 1992] Clark, J. *Windows Programmer Guide to OLE/DDE*, Prentice-Hall 1992.

[Denning 1980] Denning, P. Working Sets, Past and Present. *IEEE Transactions on Software Engineering*, 6(1): 64-84, January 1980.

[DeWitt & Carey 1984] DeWitt, D. and Carey, M. Extensible Database Systems. In *Proceedings of the 1st International Workshop on Expert Data Bases*, October 1984.

[Dyson 1992] Dyson, P. Xtensions for Xpress: Modular Software for Custom Systems. *Seybold Report on Desktop Publishing*, 6(10):1-21, June 1992.

[Graham et al. 1982] Graham, S., Kessler, P., and McKusick, M. gprof: A Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN Syympo-sium o nCompiler Construction*, pp. 120-126, June 1982.

[Hagmann 1992] Hagmann, R. Medium Term Virtual Memory Replacement. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pp. 142-147. April 1992.

[Harty & Cheriton 1992] Harty, K. and Cheriton, D.R. Application-Controlled Physical Memory Using External Page-Cache Management. In *Proceedings of the 5th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 187-197, 1992.

[Harty & Cheriton 1993] Harty, K. and Cheriton, D. R. A Market Approach to Operating System Memory Allocation. Submitted for publication, 1993.

[Kearns & Defazio 1989] Kearns, J. P. and DeFazio, S. Diversity in Database Reference Behavior. In *Performance Evaluation Review*, 1989.

[Kiczales et al. 1991] Kiczales, G., des Rivieres, J., and Bobrow, D.G. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Krueger et al. 1993] Krueger, K., Loftesness, D., Vahdat, A., and Anderson, T. Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* September 1993.

[Larus 1993] Larus, J. R. Efficient program tracing. *Computer* v26, n5. May 1993.

[Levy and Lipman 1982] Levy, H. and Lipman, P. Virtual MEmory Management in the VAX/VMS Operating System. *IEEE Computer*, pp. 35-41. March 1982.

[Martonosi et al. 1992] Martonosi, M., Gupta, A., and Anderson, T. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE 1992 Conference on Measurement and Modeling of Computer Systems*, pp. 1-12, June 1992

[McNamee & Armstrong 1990] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the First USENIX Mach Symposium.* Burlington, VT. October 1990.

[Press 1989] Press, William. *Numerical Recipes: The Art of Scientific Computing.* Cambridge, UK. Cambridge University Press. 1989.

[Rashid et al. 1988] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896-908, August 1988.

[Steele Jr. 1990] Steele Jr., G. L. *Common LISP: The Language*. Digital Press, second edition, 1990.

[Stonebraker 1981] Stonebraker, M. Operating System Support for Database Management. In *Communications of the ACM*, 1981.

[Stonebraker 1987] Stonebraker, M. Extensibility in POSTGRES. *IEEE Database Engineering*, September 1987.

[Teller & Sequin 1991] Teller, S. J. and Sequin, C. H. Visibility Preprocessing for Interactive WAlkthroughs. In *Proceedings of the 25th Annual ACM Symposium on Computer Graphics*, pp. 61-69, July 1991.

[Yokote 1992] Yokote, Y. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 414-434.* October 1992.

[Young et al. 1987] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63-76, November 1987.