Tools for the Development of Application-Specific Virtual Memory Management

Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson
Computer Science Division
University of California
Berkeley, CA 94720

April 12, 1993

Abstract

The operating system's virtual memory management policy is increasingly important to application performance because gains in processing speed are far outstripping improvements in disk latency. Indeed, large applications can gain large performance benefits from using a virtual memory policy tuned to their specific memory access patterns rather than a general policy provided by the operating system. As a result, a number of schemes have been proposed to allow for application-specific extensions to virtual memory management. These schemes have the potential to improve performance; however, to realize this performance gain, application developers must implement their own virtual memory module, a non-trivial programming task.

Current operating systems and programming tools are inadequate for developing application-specific policies. Our work combines (i) an extensible user-level virtual memory system based on a metaobject protocol with (ii) an innovative graphical performance monitor to make the task of implementing a new application-specific page replacement policy considerably simpler. The techniques presented for opening up operating system virtual memory policy to user control are general; they could be used to build application-specific implementations of other operating system policies.

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was also supported by a National Science Foundation Young Investigator Award. The authors' e-mail addresses are keithk/dloft/vahdat/tea@cs.berkeley.edu.

1 Introduction

Recently, technological advances have led to very rapid increases in many areas of computer hardware performance. Processor speed, memory and disk capacity, and network bandwidth are all improving at exponential rates. However, not all parts of computer systems are improving so quickly: because of mechanical limitations, disk latencies have not kept up with the advances in CPU speeds. These technological trends must be considered by application and operating system designers to avoid system bottlenecks whenever possible.

Most operating systems use virtual memory to provide the abstraction of large address spaces and to allow multiple large processes to remain memory resident. Because of the increasing gap between CPU speeds and disk latency times, a page fault currently takes on the order of one million instructions (clock cycles) to service. Even if CPU speeds continue to double every year, users will not see much real improvement if a constant amount of time is spent servicing page faults. Some suggest that purchasing enough physical memory that virtual memory is no longer needed; however, this solution will not work for timesharing systems or for applications whose memory usages scale with CPU speed[Hagmann 1992].

If disk access times will not be improving very quickly, it must be a priority of the operating system to use a page replacement policy which will minimize the number of page misses for a given application. Unfortunately, traditional operating systems are designed as general purpose systems intended to perform reasonably well on average over all applications. As the number of applications displaying non-standard memory access patterns increases however, a fixed operating system policy will provide optimal or near-optimal page hit-rates for fewer and fewer applications. Furthermore, because of the increasing relative cost of a single page fault, there can be marked differences in application performance between a generic page replacement policy and an optimal application-specific policy.

As a result, applications programmers that would make non-standard use of virtual memory, often bypass (or "turn off") the operating system's virtual memory system in favor of their own buffer management routines. Using knowledge of the access patterns specific to their application, they can implement a near-optimal page replacement policy. In effect, such developers are bypassing the abstractions which the operating system worked so arduously to provide. Not only is such reimplementation a difficult and time-consuming task, it violates rules of modularity and software reuse [Kiczales 1992], and further makes porting these applications to new environments much more complex than necessary.

Such shortcomings have led some to argue for user-level virtual memory management. Operating systems such as Mach [Young et al. 1987], V++ [Cheriton 1988], and Apertos [Yokote 1992] are designed to allow users to provide their own virtual memory management module[1]. All these systems, however, require users wishing to exploit these features to build or re-build a significant part of the operating system; this is a complex task beyond the ability of most users.

Our research focuses on making it easy for average programmers to develop an application-specific virtual memory management policy. We built an easily extensible user-level memory management system using a *metaobject protocol* (MOP) to facilitate user modifications to our default page replacement policy. We also built a graphical performance monitor called *VMprof* (virtual memory profiler) that allows the user to evaluate the performance of the default manager on an

[1]Apertos takes this notion one step farther: the entire operating system is *reflective*; all operating system policy can, in theory, be made application-specific.

application and compare its performance with that of other memory management policies. If a policy change is warranted, it may be quickly written using the metaobject protocol and analyzed using VMprof.

We used the combination of the MOP-based virtual memory system and VMprof to tune the page replacement policy of several applications, using an instruction-level simulator to capture their paging behavior. We describe, in a case study, how we used our techniques to optimize the performance of one application, successive over-relaxation (SOR).

The next section motivates the need for application-specific virtual memory management with a number of specific examples. Section 3 presents our protocol for page replacement policy modification in detail. Section 4 describes VMprof. Section 5 shows how our techniques are used to improve the virtual memory performance of the successive over-relaxation application. Section 6 examines some related work. Finally, section 7 presents our conclusions.

2 Background

An operating system designer may choose from a variety of page replacement policies for managing virtual memory. Most operating systems use an approximation of the least recently used (LRU) replacement policy [Levy & Lipman 1982, Denning 1980]. A large number of applications perform well under LRU, but the performance of many important applications suffers under an LRU page management policy.

A garbage collector is one application that accesses memory in a way unsuitable for LRU page-replacement [Alonso & 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 the number of garbage-collected pages is large, the application's own code and data can end up being swapped out. 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.

Database systems also have big problems under an LRU policy. They tend to access very large amounts of data in ways unexpected by LRU [Kearns & DeFazio 1989]. Typically, database management systems control their own buffer space in part to avoid using the generic policy provided by the operating system, but problems can still result if these buffers are in fact mapped into virtual memory. [Stonebraker 1981] estimates that using a page manager designed for database access patterns could improve hit rates by as much as 10-15% when dealing with such large working sets.

Some of the interactive graphics programs currently being developed [Teller & Sequin 1991] also require special virtual memory management techniques. These programs 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 often larger than physical memory, which means thrashing will occur under an LRU page replacement policy. Such applications could produce more detailed images and exhibit much higher throughput if the page replacement policy of the operating system were more finely tuned (e.g., through pre-fetching techniques).

Finally, if applications know the number of physical pages currently available to them, they can modify their runtime behavior to make optimal use of available resources [Harty & Cheriton 1992, Cheriton et al. 1991]. For example, certain Monte Carlo simulations generate a final result by averaging the results of a number of runs. Fewer runs of the simulation can be made to produce

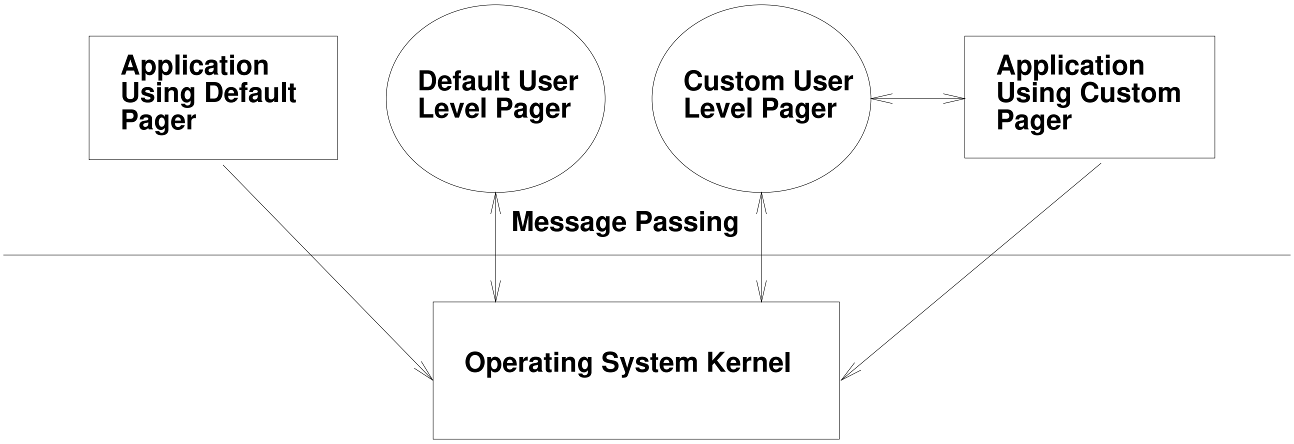

Figure 1: Operating System With Application-Specific Virtual Memory

the same results if there is a larger sample size. Such simulations could vary their sample size based on the available physical memory, thus lowering the number of page faults.

3 An Extensible User-Level Pager

Traditionally, object-oriented programming techniques have been used to simplify the task of system design for the operating system developer. To get good performance on today's computer systems however, the application programmer may also have to wear the hat of an operating system developer. In this section, we describe the extensible user-level virtual memory system we built, using an object-oriented interface to simplify the task of the application programmer. We begin with a description of the essentials of a user-level virtual memory manager.

3.1 Kernel-Pager-Application Interaction

The protocol we assumed for the interaction between the operating system kernel and the user-level virtual memory manager is based on those used for user-level pagers in Premo [McNamee & Armstrong 1990] and V++ [Harty & Cheriton 1992]. Figure 1 outlines this design. A user-level pager implements the operating system's default page replacement policy. Upon startup, the application registers its personal user-level pager with the kernel through a system call. The process's virtual memory manager then requests a number of physical memory pages from the kernel. The kernel responds by allocating as many physical pages as possible for that process. These physical pages are used as a cache for the process's virtual address space. The operating system may dynamically allocate and deallocate physical pages to a running process in response to memory access patterns and the need of other processes in the system. An efficient implementation of such a dynamic resource allocation scheme can be found in [Anderson et al. 1992].

The necessary communication between the application, the operating system, and the user-level pager to respond to a page fault is summarized as follows:

- Upon a page fault, a trap into the operating system takes place. The kernel makes an upcall to the application's pager with the faulting virtual address. This upcall dispatches the procedure `HandlePageFault` (detailed below) within the user level pager. This procedure is responsible for bringing the missing page into physical memory (through system calls back into the operating system).

- If all of the process's physical pages have been allocated, the pager is responsible for choosing a page to replace. The user-level pager polls the operating system to obtain information (for example, hardware page usage and modified bits) regarding a process's page access patterns. This information is then used to choose the page to replace.
- If the page chosen for replacement has been modified, it must be written to the backing store. This is done through system calls into the operating system.
- If an application needs to change its virtual memory policy during its execution, a communication channel is established between the user-level pager and the application to communicate such needs.
- The operating system makes an upcall to the pager to inform it of changes in the number of available physical pages.
- The user-level pager can 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. For instance, this capability could be used to implement distributed virtual memory [Appel & Li 1991].

3.2 A Protocol for Extensibility

We now describe our object-oriented user-level virtual memory manager. Our goal was two-fold. First, we wanted to provide the standard parts of a virtual memory system that were unlikely to change for different policies. A fair amount of the code for traditional virtual memory systems is taken up with bookkeeping and other infrastructure. Second, we wanted to expose the key elements of the system's policy decisions to user change. We did this by structuring the system in an object-oriented fashion, allowing programmers to tweak our code by building derived objects that change only the parts of our implementation that truly needed to be changed. In addition, by using an object-oriented approach, we can allow multiple policies to exist for the *same* application, for instance, for different areas of memory and for different phases of the program's execution. In all this, we were guided by the principles of metaobject protocol design—how to design an interface to allow the system to be easily customizable by it users.

The protocol allows for memory management on a per address space basis; at a high-level the protocol consists of the following classes and methods:

- The class `AddressSpace` encapsulates the physical memory available to the process, the process's page table, and all state information necessary to implement the desired paging policy
- The method `HandlePageFault` is called through an upcall from the kernel whenever a page fault occurs.
- The method `FindPageToReplace` is called by `HandlePageFault` to select a page for removal.
- The method `PollKernel` is called periodically to retrieve the state information (most often, page usage bits) necessary to implement the desired paging policy.

Our user-level memory manager is a simple one: it implements an approximation of LRU and ignores such issues as shared memory segments and sparse address spaces[2]. The following C++[3] classes outline our protocol:

```
class CoreMapEntry {
    int  physicalFrame;
    PTE  *virtualPage;
    Bool free;
};

class CoreMap {
    void         AddPageToAllocation(int physicalFrame);
    void         RemovePageFromAllocation(int physicalFrame);
    int          FindFreePage(); // Calls FindPageToReplace if none available
    int          GetNumFreePages();
    List         *allocatedPages; // List of CoreMapEntry
    int          tableSize;
    List         *freePages; // List of CoreMapEntry
};
```

There is an instance of `CoreMapEntry` for each physical page allocated to a process. The member `physicalFrame` is the number of the page in system memory; it is used as a tag for communication between the pager and the kernel. The `CoreMap` class is a list of physical pages allotted to a particular process. The list of pages available to a process may change dynamically; `AddPageToAllocation` and `RemovePageFromAllocation` are called by the kernel (via an upcall) to notify the user-level system of these changes.

```
class PTE { /* Page Table Entry */
    CoreMapEntry *physicalFrame;
    Bool         valid;
    Bool         modified;
    Bool         used;
};

class AddressSpace {
    virtual int  FindPageToReplace();
    virtual void HandlePageFault(int faultPage);
    virtual void PageFetch(int pageToReplace, int faultPage);
    virtual void PollKernel(...); // Get page usage information
    PTE     *pageTable;
    int     pageTableSize;
    CoreMap physicalMemory;
    int     LRUClockHand;
};
```

An `AddressSpace` encapsulates both a process's virtual memory and state information for its page replacement policy. Each page of virtual memory has a PTE. The pager periodically calls

[2] We are currently working on extending the manager to accomodate these additional features

[3] For brevity's sake, we do not include accessor functions, constructors, or destructors, nor do we distinguish between public and private members.

PollKernel to retrieve the process's recent page access patterns. PageIn reads faultPage from the backing store and stores it in pageToReplace, writing pageToReplace to the backing store if modified.

```
void
AddressSpace::HandlePageFault(int faultPage)
{
  int pageToReplace;

  if (physicalMemory.GetNumFreePages() == 0)
    pageToSwap = FindPageToSwapOut();
  else
    pageToSwap = physicalMemory.FindFreePage();
  PageIn(pageToSwap, faultPage);
}

// Implementation of a one-bit clock algorithm approximating LRU
int
AddressSpace::FindPageToReplace()
{
  int  pageFound = 0;

  while (!pageFound) {
    LRUClockHand++;
    if (pageTable[LRUClockHand].valid)
      if (!pageTable[LRUClockHand].used)
        return LRUClockHand;
      else
        pageTable[LRUClockHand].used = FALSE;
    if (LRUClockHand == pageTableSize)
      LRUClockHand = 0;
  }
}
```

Given that the above classes and methods implement the default policy, we now demonstrate how our implementation can be specialized. If a programmer decides that a Most Recently Used (MRU) page replacement policy is more appropriate for his application, as might be the case if the application were scanningly linearly through a large data structure, the following definitions could be added to the default pager:

```
class MRUAddressSpace
     : public AddressSpace {
  public:
    void FindPageToReplace(); // Override base class definition
  private:
    int  MRUClockHand;
};

// Assumes at least one page referenced since last fault.  One
// bit approximation of MRU.
```

```
int
MRUAddressSpace::FindPageToReplace()
{
  while (!pageFound) {
    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 on `FindPageToReplace` is now called when a page fault occurs in a process running in an `MRUAddressSpace`. In order to implement an MRU policy through the protocol, the programmer simply created one new class and modified one documented function. The details of other functions and classes were unchanged. In a similar vein, `HandlePageFault` could be modified to allow for pinning and pre-fetching of pages. In summary, our design using a metaobject protocol provides a clean interface for *disciplined* modification of the page replacement policy.

4 VMprof – The Virtual Memory Profiler

Quickly developing application-specific operating system policies requires more than an extensible implementation of these policies. It is difficult to predict the dynamic behavior of a particular application/policy combination, and therefore difficult for a programmer to judge how well modifications to the operating system are working. We suggest that the MOP operating system design methodology be extended with visual tools to display the dynamic behavior of the operating system. We designed a visual performance tool, VMprof, that allows the programmer to easily identify problems with virtual memory performance. The MOP-based virtual memory manager and VMprof complement each other and increase the speed and accuracy of virtual memory policy development.

VMprof supplements program analysis tools such as UNIX gprof[Graham et al. 1982], and MemSpy[Martonosi et al. 1992]. Given a trace of page faults, VMprof allows the user to analyze both spatial and temporal aspects of virtual memory management. VMProf's 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 policy behavior.

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

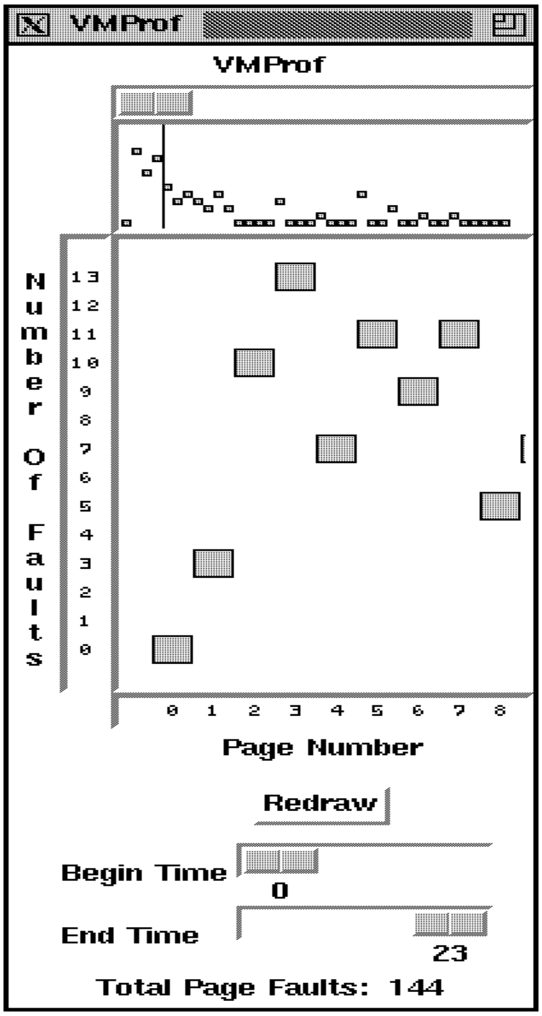

Figure 2: The VMprof Graphical Display

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 virtual memory 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. If a policy change is warranted, it may be quickly written using the metaobject protocol and analyzed using VMprof.

5 SOR – A Case Study

We now describe in detail how our techniques were used to tune virtual memory performance in a particular case, successive over-relaxation (SOR). In this application, each element of a matrix is averaged with its four immediate neighbors. This operation is repeated on the matrix until a steady state for the matrix's values is reached. The following code fragment is a simplification of SOR[4]:

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

We simulated SOR to generate a trace of memory accesses. We then fed these accesses into each proposed virtual memory page replacement policy to identify the page faults in the trace. These page faults finally fed into VMprof to provide a profile of the performance of SOR on each policy.

To keep our simulations tractable, we chose a relatively small matrix size and a correspondingly small physical memory size—the matrix occupies 512 pages, one page is needed for the application's code. In this example, we will assume that 512 pages of physical memory are available to the application. Similar paging behavior would have been observed for larger matrices and correspondingly larger physical memory. We finally assumed that each row of the matrix occupied exactly two pages of memory.

The results of running SOR with a LRU page replacement policy[5] can be seen in Figure 3. The row labeled *faults* indicates that page faults are frequent: there is one fault per iteration of the loop per page of data when the size of physical memory is just less than virtual memory. The user watching the number of page faults updated with respect to time sees a continuous left-to-right cascading of faults. This suggests that thrashing is occurring.

Clearly, the performance of LRU is inadequate under the given circumstances. The user can achieve much better performance by studying SOR's particular access patterns in a little more detail. The access pattern for SOR can be seen graphically in Figure 4. On the first iteration of the outer loop, there have to be 513 page faults, since none of the pages have been previously accessed; this is independent of the page replacement policy (unless pre-fetching techniques are employed). On subsequent iterations of the outer loop, however, the page faults that occur depend on the virtual memory policy. If calculating the value for `matrix[k][j]` causes a page fault in reading `matrix[k][j+1]`, then an LRU policy will choose the physical page which has not been used for the longest period of time (page $n+6$ in Figure 4). Unfortunately, given the cyclic sequential access to memory, 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, there is a page fault on every page of data of the matrix for each iteration through the loop for LRU.

An ideal page replacement policy replaces the page which will not be needed for the longest time into the future. If reading `matrix[k][j+1]` causes a page fault, then the page which will not be referenced for the longest time is the first full page immediately before `matrix[k][j-1]`. Thus, a custom page handler should choose virtual page n from Figure 4 for replacement. A custom page replacement policy to effect this algorithm could be written fairly quickly through the MOP by writing a new version of `FindPageToReplace`. The 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

[4]The code for SOR could be restructured to make its memory accesses more local; such rewriting, however, is often non-intuitive or even impossible.

[5]A 1-bit clock algorithm was used to approximate LRU.

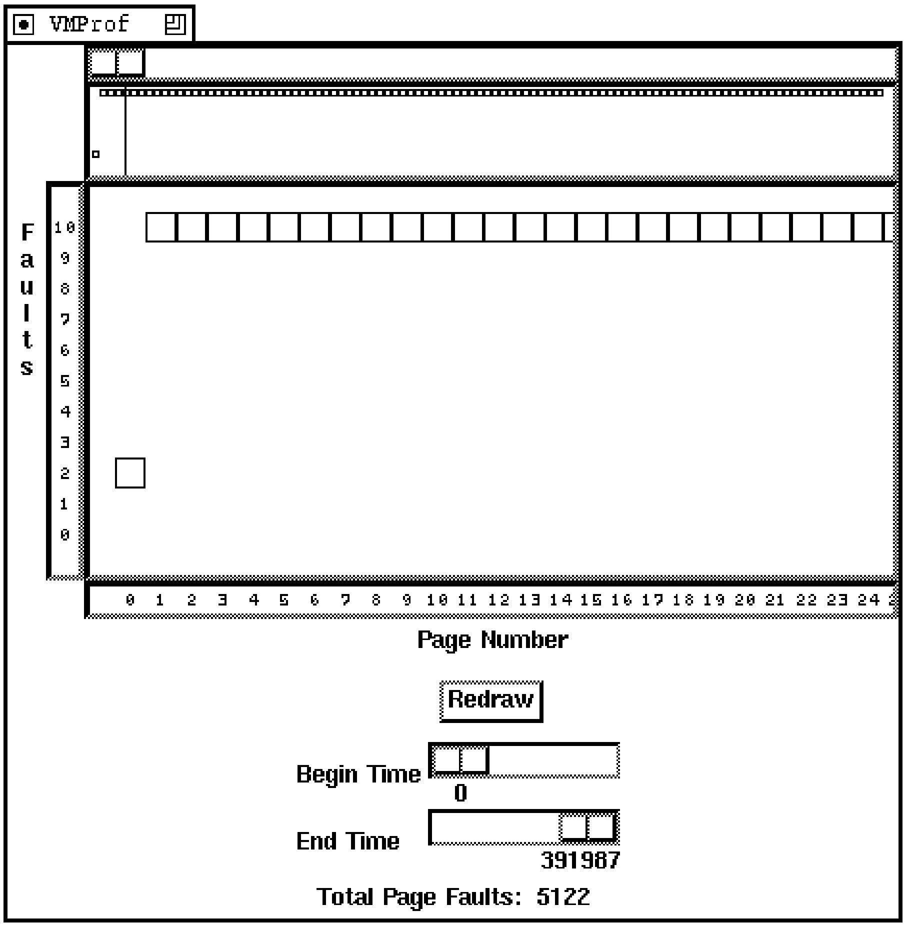

Figure 3: SOR with a LRU Page Replacement Policy.

page v, then the ideal page to replace is the physical page containing virtual page $v-5$. The custom policy has the flavor of Most Recently Used (MRU) page replacement policy; however, MRU does not perform well in this case because the MRU page is almost certain to still be needed.

We implemented such a replacement policy in our virtual memory simulator by slightly modifying earlier code we had written for an MRU page replacement policy. Running the resulting traces through VMprof, we obtain the results in Figure 5. Using this policy, after the initial "startup-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 reduction in the number of page faults from 5121 for LRU to 522 for the custom policy. Clearly, the amount of the advantage depends on the difference between virtual memory size and the available physical memory.

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 differential between available physical memory and required virtual memory (y-axis). From the plot, we see that the number of page faults for the custom policy is dependent only on the number of iterations through the loop and the differential. The performance of the LRU policy is uniformly poor, independent of the number of available physical pages. LRU provides an upper bound for the number of page faults; as the number of available physical pages decreases, the customized policy's performance begins to approach that of

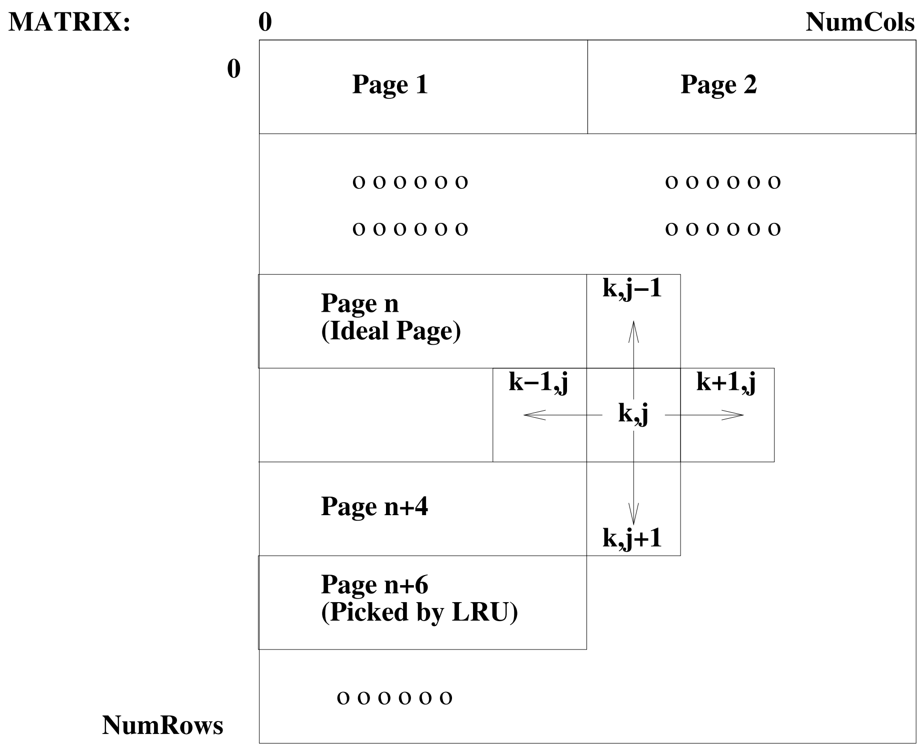

Figure 4: A graphical description of the entries which SOR accesses in calculating the value for `matrix[k][j]`. Access to `matrix[k][j+1]` on page $n+5$ causes a page fault.

LRU.

The above tuning of virtual memory management did not exploit two very important techniques used to improve page hit rates: page pinning and pre-fetching. With page pinning, an application developer may realize that certain pages within the application should not be paged out even if they otherwise meet the criterion for page replacement. A likely candidate for pinning is the code of a garbage-collected program: the code pages of the application should be pinned so that data pages being collected will always be paged out before the application's code.

Pre-fetching is useful when an application is aware that a large number of pages will be accessed in sequence. Rather than having a fault on each new page access, pages can be brought into physical memory before they are needed. Clearly, prefetching would be very useful in the SOR example described above. Rather than having a page fault on every new page access during the first iteration of the loop, the application could request that the user-level pager prefetch ahead some number of pages. The application would still be limited by disk bandwidth, but it would incur less fault overhead. It is important to keep in mind that while pre-fetching can often times vastly reduce the number of page faults, page hit rates for an application employing pre-fetching cannot be directly compared against the same application which does not use pre-fetching.

In summary, the following steps are typically needed to tune a page replacement policy:

- Identify and isolate phase transitions in program using the visual cues provided by the performance tool.

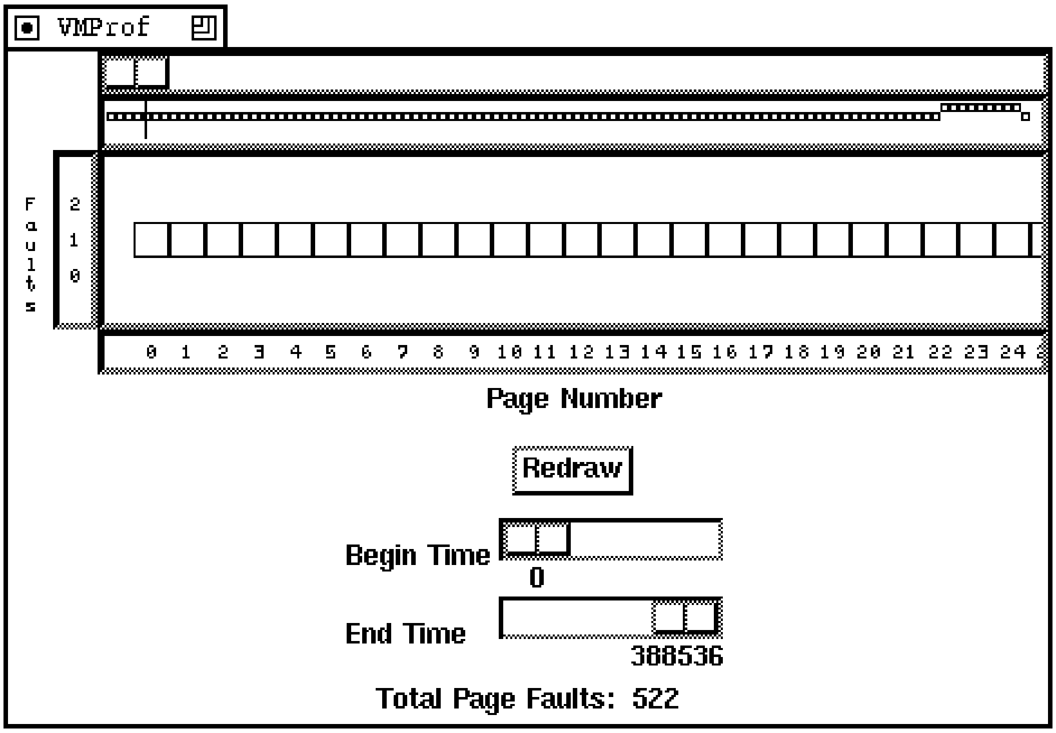

Figure 5: SOR with a Customized Page Replacement Policy.

- Experiment with different page-replacement policies to see whether one is appropriate for the measured access pattern.
- If a combination of policies or an entirely new policy is desired, write a custom manager using the metaobject protocol and evaluate it.

The methods described above can be used in combination with one another to switch policies on the fly through 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.

6 Related Work

Our approach to building an extensible user-level pagers exploits the idea of a metaobject protocol. Metaobject protocols were originally developed as a technology to open up programming language implementation [Kiczales et al. 1991, Kiczales et al. 1992, Vahdat 1993]. Compilers for high-level programming languages must make a number of decisions about lower-level implementation details. These decisions are not always appropriate for all applications. A metaobject protocol for the Common Lisp Object System (CLOS) [Steele Jr. 1990] was designed to allow programmers to modify decisions and assumptions made by the compiler. In this way, the performance of the code produced by their compiler could be tuned to compete with such low level programming languages as C.

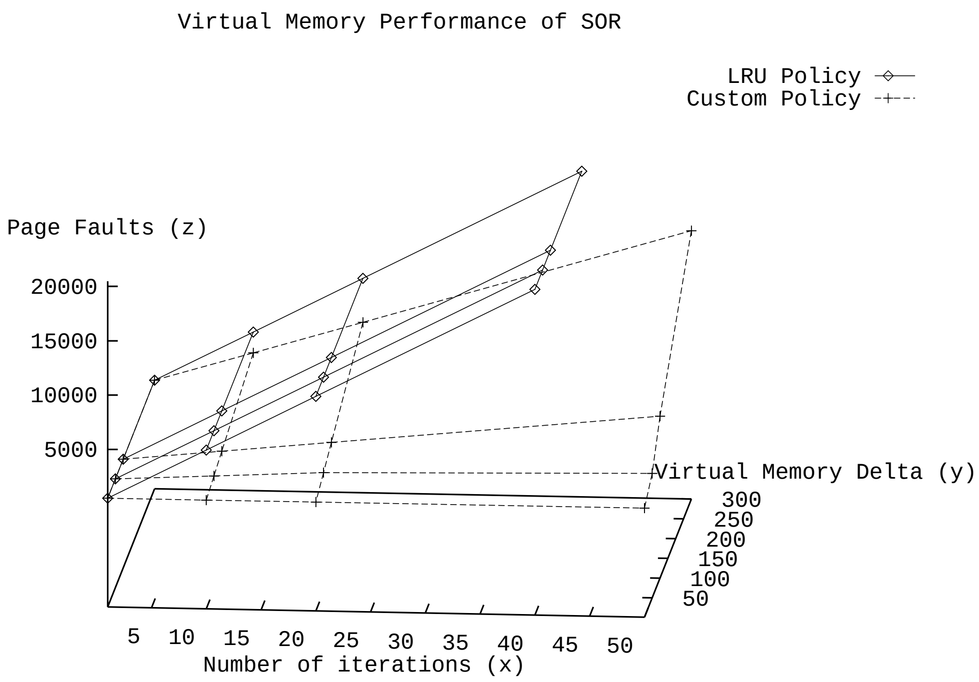

Figure 6: Number of page faults with SOR.

We chose MOPs to open up the virtual memory system because of the clear parallel between performance issues in compiler design and the operating system. The MOP design philosophy dictates that our protocol should observe the following considerations:

- *Scope control:* Changes made in one area of the operating system should not affect logically separate modules. For example, changes made to the virtual memory system should not affect the communications package. Furthermore, changes made for one application should not affect the behavior of other applications.
- *Orthogonality:* It should be possible to use the metaobject protocol to customize particular aspects of the implementation without having to understand it in its entirity.
- *Incrementality:* The user should not have to reimplement an entire module if only a slight modification is required [Kiczales & Lamping 1992]. The functions of a given part of the particular module should be clearly documented to allow for such changes.
- *Safety:* Bugs in a programmer's metacode should have only a limited effect on the rest of the system.

In the case of virtual memory management, scope control and safety are provided simply by moving the virtual memory policy to user level. A user can configure a new policy without changing the policy used for any other application. Beyond that, we structured our implementation to satisfy the desire for orthogonality and incrementality.

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. Perhaps most analogous to our work, 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.

7 Conclusions

In conclusion, the techniques we describe allow users to experiment with various page replacement policies and to get immediate feedback from the user interface. This feedback may be used to choose another paging policy, to design one which inherits features from another policy, or to develop a unique policy that meets an application's demands. The metaobject protocol allows for user modification of a complex system, and the user interface provides a facility to evaluate the different tradeoffs involved with modifying operating system policy.

8 Acknowledgements

We wish to thank Gregor Kiczales, John Lamping, and Luis Rodriguez for helping to explain the metaobject protocol design approach. Paul Hilfinger assisted in ironing out some of the object-oriented programming issues. Douglas Ghormley, Gregor Kiczales and John Lamping provided many helpful comments on drafts of this paper.

References

[Alonso & Appel 1990] Alonso, R. and Appel, A. 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*, 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.

[Cheriton 1988] Cheriton, D. R. The V Distributed System. In *Communications of the ACM*, pp. 314–333, March 1988.

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

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

[Graham et al. 1982] Graham, S., Kessler, P., and McKusick, M. gprof: A Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 120–126, June 1982.

[Hagmann 1992] Hagmann, R. Medium Term Virtual Memory Replacement. In *Proceeding 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*, 1992.

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

[Kiczales & Lamping 1992] Kiczales, G. and Lamping, J. Issues in the Design and Documentation of Class Libraries. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1992.

[Kiczales 1992] Kiczales, G. Towards A New Model of Abstraction in Software Engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

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

[Kiczales et al. 1992] Kiczales, G., Ashley, M., Rodriguez, L., Vahdat, A., and Bobrow, D. G. Metaobject Protocols — Why We Want Them and What Else They Can Do. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1992.

[Levy & 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 '92 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 Accomodate User-Level Page Replacement Policies. In *Proceedings of First USENIX Mach Symposium*, Burlington, Vermont, October 1990.

[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.

[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.

[Vahdat 1993] Vahdat, A. The Deisgn of a Metaobject Protocol Controlling the Behavior of a Scheme Interpreter. Technical report, Xerox PARC, March 1993.

[Yokote 1992] Yokote, Y. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 414–434. ACM, 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.