# 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

**Abstract**

While many applications incur few page faults, some scientific and database applications perform poorly when running on top of a traditional virtual memory implementation. To help address this problem, several systems have been built to allow each program the flexibility to use its own *application-specific* page replacement policy, in place of the generic policy provided by the operating system. This has the potential to improve performance for the class of applications limited by virtual memory behavior; however, to realize this performance gain, application developers must re-implement much of the virtual memory system, a non-trivial programming task.

Our goal is to make it easy for programmers to develop new application-specific page replacement policies. To do this, we have implemented (i) an extensible object-oriented user-level virtual memory system and (ii) a graphical performance monitor for virtual memory behavior. Together, these help the user to identify problems with an application's existing paging policy and to quickly modify the system to fix these problems. We have used our tools for tuning the virtual memory performance of several applications; we present one case study illustrating the benefits and the limitations of our tools.

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation, the Xerox Corporation, 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

Recent technological advances have led to very rapid increases in many areas of computer hardware performance. Processor speed, network bandwidth, memory and disk capacity 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. On current hardware, a single disk read can take the equivalent of about one million machine instructions to service, and this gap is likely to increase in the future.

Operating systems have traditionally used virtual memory to help hide the gap between CPU speed and disk access time [Denning 1980]. Virtual memory provides the illusion of a large, fast address space for each application, by managing physical memory as a cache for disk. To exploit the temporal and spatial locality in the memory access patterns of most programs, operating systems commonly use an approximation of "least recently used" (LRU) [Levy & Lipman 1982] as the page replacement policy. This has been remarkably successful: applications can be written independent of the amount of physical memory available to each job, yet the overhead associated with virtual memory is not an issue for most programs.

However, some applications can perform extremely poorly when running on top of virtual memory — examples include scientific applications, databases, garbage collected systems, and graphics

programs. These applications all have memory requirements that can exceed the amount of physical memory on the machine, but they do not display the locality needed to maintain the illusion that access to virtual memory is as fast as running directly on top of physical memory. Worse, this problem will not improve with time: the performance of these applications will continue to deteriorate as the relative performance of disks continues to get slower.

One possible solution would be to try to develop a "hero" virtual memory system that performs well for these programs (see [Hagmann 1992] for an attempt). The difficulty in developing such a system is that the operating system page replacement policy must balance 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 LRU, and thus hurt overall system performance.

Instead, we consider a different approach, one which offers near-optimal performance for all programs. A number of systems have been built which allow each application to specify its own *application-specific* virtual memory policy. The operating system kernel is responsible for allocating the machine's 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 applications can use a policy tuned to their specific needs. Mach [Young et al. 1987, Rashid et al. 1988], V++ [Harty & Cheriton 1992], and Apertos [Yokote 1992] are all systems that implement this approach.[1]

While these systems have the potential for large improvements in application performance, it can be difficult for an ordinary application programmer to realize the potential performance gain. The programmer must be able to diagnose the problem with the default page replacement policy and then, to change the policy, implement a new virtual memory system.

In this paper, we describe a toolkit we have built to make it easy for programmers to develop new application-specific page replacement policies. First, we have implemented an *extensible* user-level virtual memory system; this system is object-oriented, with disciplined entry points for non-expert programmers to easily modify key policy choices. Second, we have developed *VMprof*, a graphical debugging tool to allow a user to evaluate competing policies. We have used the the combination of our extensible 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 a case study of one of these applications, successive over-relaxation (SOR). This example illustrates a key advantage of custom policies: graceful degradation of performance when the system would otherwise start to thrash.

The remainder of this paper discusses these issues in more detail. In the next section, we motivate the need for application-specific virtual memory management. Section 3 describes the kernel support we need, while Sections 4 and 5 describe our toolkit. Section 6 presents our case study. Section 7 discusses our work in the context of related work, and Section 8 summarizes our conclusions.

# 2 Motivation

In this section, we motivate the need for application-specific virtual memory management by first describing some common applications which do not perform well under LRU, and then outlining some of the problems with alternative solutions.

## 2.1 Examples

A database is the canonical example of a system which performs poorly under an LRU pol-

[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.

icy [Kearns & DeFazio 1989]. Databases often scan through large amounts of data in a sequential or even random fashion. While the code implementing the database should be managed with an LRU policy, the ideal policy for the data changes from operation to operation. Worse, the database is often aware of its access patterns ahead of time, but it has no way of informing the operating system of its needs. 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 performance by an order of magnitude.

A garbage collector is another application that accesses memory in a way unsuitable for LRU [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. Further, the memory allocator should not reuse parts of the heap that have been swapped out until there is room for those pages to be swapped back in.

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 based on the semantics of the program).

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 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. Adapting application behavior to the amount of available physical memory is straightforward if there is user-level control over virtual memory, but nearly impossible if paging is implemented in the kernel, transparent to the application.

## 2.2 Alternative Solutions

Before discussing how to build an application-specific virtual memory system, we consider some alternatives. One would be to devise an ideal page replacement policy which performs well for all of the above applications, as well as for programs that do well under an LRU policy. Such a policy does not exist today, and moreover, we believe it is unlikely to in the near future. Because of the large gap between CPU and disk performance, even if using a generic policy results in only a few extra page faults relative to using an optimal application-specific policy, there can be a large impact on application performance. In the absence of an ideal paging system, application programmers have been faced with the following options, none of which are always applicable:

- Purchase enough memory so that the application fits in physical memory. While this may appear attractive because memory is increasingly inexpensive, application programmers often would like to scale to even larger data set sizes. So, irrespective of the amount of physical memory, there will always be problems which require more memory than the

hardware can support [Hagmann 1992].

- Bypass the operating system's virtual memory system by pinning a pool of the application's pages into physical memory [Stonebraker 1981]. User code explicitly manages the buffer pool as a cache for disk by deciding which disk pages get swapped into main memory. This can require large changes to application code since accesses to data structures must now indirect through the buffer pool manager, and it is inflexible in a multiprogrammed environment [Harty & Cheriton 1993]. Worse, this essentially requires each application programmer to re-implement the virtual memory system. Our tools allow a separation of concerns: the application can be written in terms of normal memory reads and writes, while the paging policy can be easily changed with no changes to the application code.

- Restructure the application to improve its spatial and temporal locality. This technique, known as "blocking", is commonly used within the scientific programming community to improve processor cache performance, but it can also be used to improve virtual memory behavior. While the result can be obscure code bearing little resemblance to the original program, a blocked program can have much better performance. Our monitoring tool, VMprof, helps identify the places in the application code where blocking would benefit performance; moreover, even after an application is re-written, there may still be a benefit to using a custom paging policy in place of LRU. The example SOR application described in Section 6 illustrates the advantage of using our tools on a blocked program.

# 3 Kernel Support for User-Level Virtual Memory Management

Before describing our extensible virtual memory manager, we first outline the kernel support necessary to allow each application to set its own paging policy at user level. Support similar to what we describe here is provided by Mach (as extended by [McNamee & Armstrong 1990], V++, and Apertos.

The key observation is to appropriately divide responsibility for virtual memory between the application, the kernel, and a separate user-level paging system, specific to the application. This organization can be seen in Figure 1. In the simplest case, the application sees no change: it can do normal reads and writes to its virtual memory locations, and the combination of the kernel and the user-level pager handle any page faults that occur, completely transparently to the application.

The operating system kernel is modified to hand off control over paging policy to the user level. Unlike a traditional 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. For reasons of security, page tables cannot be modified directly at the user level. Each user-level manager is given a set of physical pages to manage by the kernel, and has complete control over which of the application's virtual pages are to be assigned to those physical pages and which are to be on disk. In this way, the *mechanisms* necessary to implement virtual memory are separated from the application-specific *policy* implemented at the user level.

Whenever the kernel would make a virtual memory policy decision, the kernel makes an upcall to the user-level pager instead.[2] For instance, on a page fault, a decision is needed as to which page currently in memory will be swapped to disk to make room for the incoming page. Instead of making this decision itself, the kernel upcalls to the

[2] An upcall is the reverse of a system call. A system call implements a procedure call from application code to a kernel routine, while an upcall implements a procedure call from the kernel to application code.

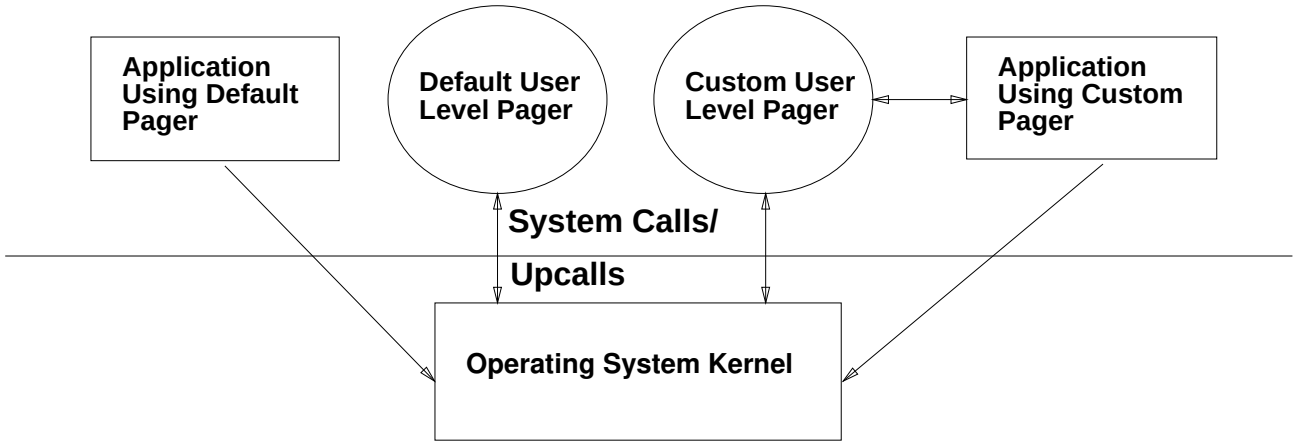


Figure 1: Operating System With Application-Specific Virtual Memory

user-level pager on each fault, providing the needed information (such as hardware page reference information) to the user-level pager. The user-level pager then chooses which page to replace. The kernel is also responsible for informing the user-level pager of changes in the number of pages assigned to it.

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.

The interactions between the application, the operating system kernel, and the user-level pager are 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 a procedure `HandlePageFault` 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, or if it needs to know which of its pages are mapped, 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. A system call is added to inform the kernel that the application needs more/fewer pages of physical memory [Harty & Cheriton 1993].

- 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 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].

Moreover, there is not a one-to-one mapping between applications and paging policies. The same default user-level pager can be used by the majority of applications that perform acceptably well with an LRU paging policy. By contrast, a single application might have multiple policies, one for each segment of memory and/or one for each phase of its execution.

# 4 An Extensible User-Level Pager

In this section, we describe our extensible user-level paging system. We had two goals. 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 [Bershad et al. 1988]. 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 the interest of brevity, we focus on the parts of our system that an application programmer might need to understand in order to implement a custom page replacement policy. We leave out those details needed to deal with sharing of memory segments, sparse address spaces, portability, page coloring, etc. We refer readers to [Young et al. 1987] for a more complete description of the issues in building a robust, general-purpose virtual memory system.

The simplified protocol we discuss here allows for memory management on a per process basis; this can be easily extended to allow paging policy to be set on a per memory segment basis. At a high level, the protocol consists of the following classes and methods:

- A `ResidentPageTable` object is associated with each process. It encapsulates information about the physical pages assigned to the process by the kernel, for instance, the virtual page contained in the physical page, if any. By default physical pages are divided into three lists: a free list of all unallocated pages, a list of all deallocated pages which have not yet been unmapped (providing a sort of "second-chance" cache [Levy & Lipman 1982]), and a list of all mapped pages. The relative size of these lists can be controlled by the user, whereas the total number of pages assigned to the process is controlled by the kernel.

- An `AddressMap` object also is associated with each process. It encapsulates information about the process' virtual pages, for instance, the physical page containing the virtual page, if the page is in memory, and any hardware reference information. In other words, the address map is similar to a traditional page table.

- The method `HandlePageFault` on `AddressMap` 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 to retrieve the state information (most often, page usage bits) necessary to implement the desired paging policy.

The following C++[3] classes outline the interface to our implementation:

[3] For brevity's sake, we do not include accessor functions, constructors, or destructors.

```
class RPTE { // Resident Page Table Entry
  public:
    int  physicalFrame;
    // Address Map Entry
    AME  *virtualPage;
    Bool free;
};

class ResidentPageTable {
  public:
    // Upcalls from the kernel
    void AddPageToAllocation(int pageNum);
    void RemovePageFromAllocation(int pgNum);
    // Calls FindPageToReplace if none avail
    int  FindFreePage();
    int  GetNumFreePages();
  private:
    int  tableSize;
    // Following are lists of RPTE
    List *allocatedPages;
    List *unmappedPages;
    List *freePages;
};
```

There is an instance of `RPTE` (resident page table entry) 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 `ResidentPageTable` 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 AME { // Address Map Entry
  public:
    RPTE  *physicalFrame;
    Bool  valid;
    Bool  modified;
    Bool  used;
};

class AddressMap {
  public:
    virtual int  FindPageToReplace();
    virtual void HandlePageFault(int faultPg);
    virtual void
      FetchPage(int targPage, int faultPage);
    // Get page usage information
    virtual void PollKernel(...);
  private:
    AME               *pageTable;
    int               pageTableSize;
    ResidentPageTable coremap;
    int               LRUClockHand;
};
```

An `AddressMap` encapsulates both a process's virtual memory and state information for its page replacement policy. Each page of virtual memory has a AME (Address Map Entry). The pager periodically calls `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.

In the simplest case, the user-level virtual memory manager consists of instances of an `AddressMap` and a `ResidentPageTable` object. A programmer can create a new paging policy by changing the methods for these objects, compiling a new memory manager and asking the kernel to use the new manager for the application. Multiple policies can also be coded into the same manager so that applications can change their page replacement policy "on the fly" (as their memory access patterns change). The following methods are provided with the memory managers and implement the default approximation to LRU.

```
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[LRUClockHand].used = FALSE;
    if (LRUClockHand == pageTableSize)
      LRUClockHand = 0;
  }
}
```

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 scans linearly through a large data structure, the following definitions could be added to the default pager:

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

// Assumes at least one page referenced since
// last fault.  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 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 order to provide multiple page replacement policies for different memory segments of a process (as is necessary for databases: a different policy is needed to manage the code as opposed to the data), the user would simply create a subclass of the `ResidentPageTable` which breaks up the allocated physical pages into lists corresponding to the process's different logical segments. Then, `HandlePageFault` determines which segment caused the fault and calls an appropriate specialization of `FindPageToReplace`. For databases, an LRU policy might be used for the code segment, while an MRU-like policy might be optimal for the data segment if the process is scanning through large amounts of data. Thus, the user-level memory manager provides very fine-grained control over the desired page replacement policy based both the faulting segment and the process's current access patterns.

# 5 VMprof – The Virtual Memory Profiler

Given the complex tradeoffs 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 identify performance problems with the particular application/policy combination, to help identify ways to improve performance. Further, the user needs to be able to quickly evaluate the performance effect of changes to the application and/or the paging policy.

To address this, we have designed a visual performance tool, VMprof, to allow the programmer to easily identify problems with virtual memory performance by displaying the dynamic behavior of the paging system. The user-extensible virtual memory manager and VMprof complement each

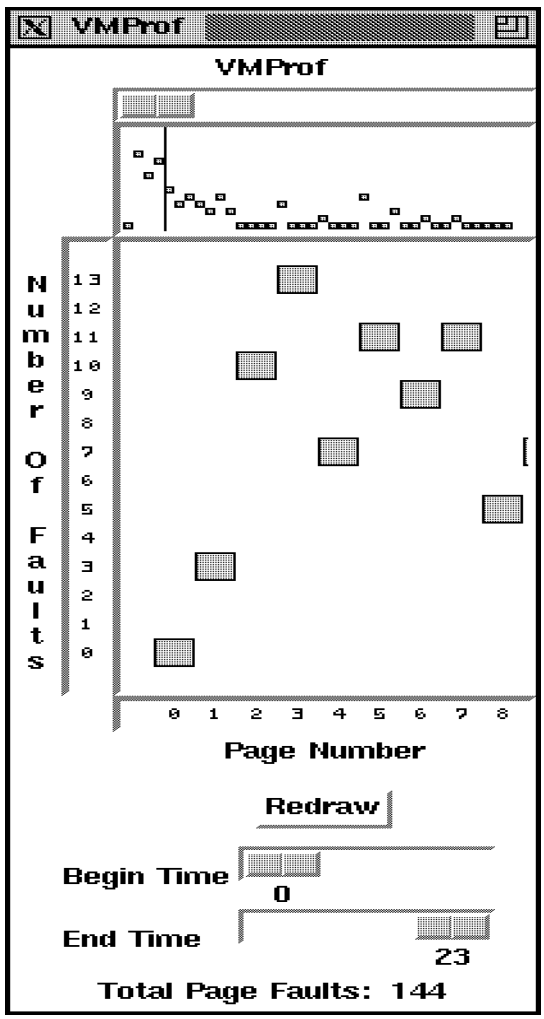


Figure 2: The VMprof Graphical Display

other by decreasing the time needed to tune virtual memory performance.

VMprof supplements other program performance 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 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.

Based on experience using VMprof, we have identified 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. 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 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 may 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 SOR – A Case Study

We now describe a case study of how our techniques were used to tune the virtual memory performance of an implementation of successive over-relaxation (SOR). We generated memory reference traces for a basic implementation of the SOR application for several different problem sizes; we then determined the application's paging behavior 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 to VMprof to allow us 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:

```
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;
```

To illustrate the process of tuning virtual memory behavior, Figure 3 displays the output of the VMprof tool, profiling our initial SOR implementation with an LRU page replacement policy.[4] The matrix size was chosen to be 1K by 1K, with each element being a double precision floating point number; in other words, the matrix consumed 8 megabytes of virtual memory. To illustrate paging behavior, we assumed there were 8 megabytes (2048 4KByte pages) of physical memory available. Since the application code takes a small but non-zero amount of space, the program barely does not fit in physical memory.

As shown in Figure 3, the cyclic access pattern of SOR, combined with LRU, results in a large number of faults. The row labeled *faults* indicates that

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

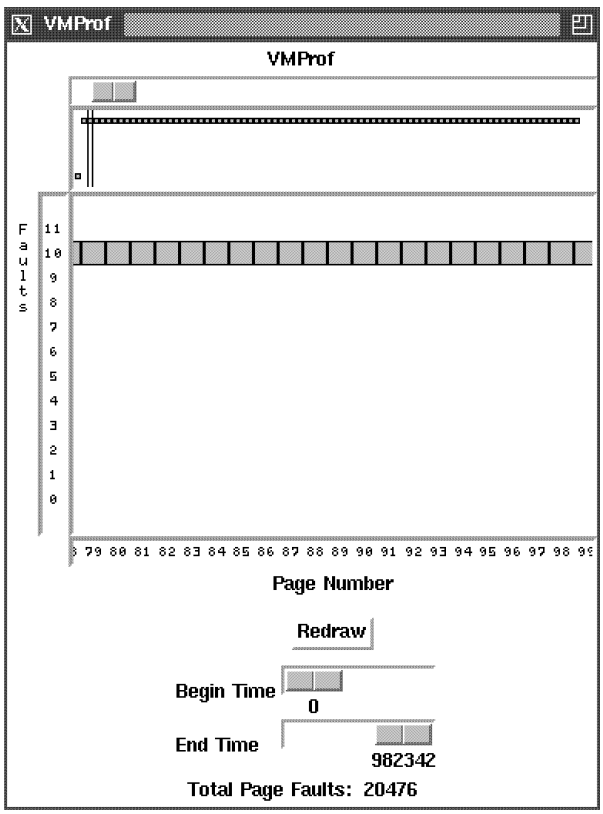


Figure 3: SOR with a LRU Page Replacement Policy

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. This suggests that thrashing is occurring.

The performance of LRU is much worse than optimal in this situation. The access pattern for SOR can be seen graphically in Figure 4. On the first iteration of the outer loop, there are 2K compulsory 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 number of page faults does depend on the virtual memory policy. If calculating the value for `matrix[r][c]` causes a page fault in reading `matrix[r+1][c]`, 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, assuming each row of the matrix takes 2 pages of memory). 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, 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. If reading `matrix[r+1][c]` causes a page fault, then the page which will not be ref-

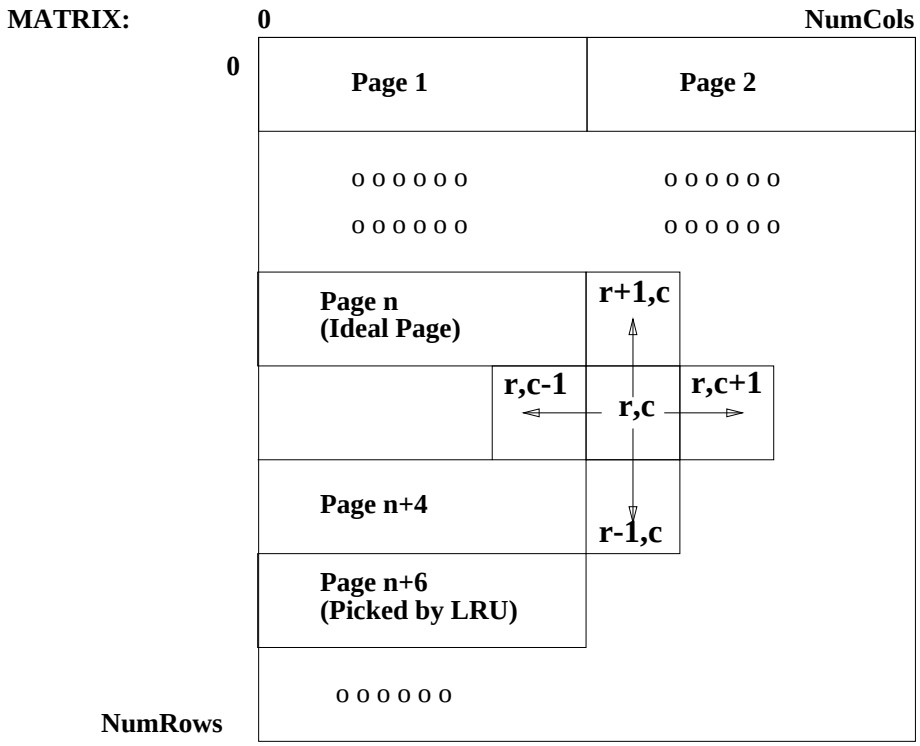


Figure 4: SOR Access Pattern – access to `matrix[r+1][c]` on page $n + 5$ causes a page fault.

erenced for the longest time is the first full page immediately before `matrix[r-1][c]`. Thus, a custom page handler should choose virtual page $n$ from Figure 4 for replacement. A custom page replacement policy to effect this algorithm may be quickly created 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 page $v$, then the ideal page to replace is the physical 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; however, MRU does not perform well in this case because the MRU page is almost certain to still be needed.

We implemented this custom replacement policy by slightly modifying the implementation of MRU described in Section 4. We then re-ran the address trace through our simulator, invoking the new paging policy. The resulting VMprof output is displayed in Figure 5. Using this policy, after the initial 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.

However, the amount of the advantage of our custom paging policy relative to LRU depends on the difference between the virtual memory needed and the available physical memory. With LRU, we thrash as soon as we need more virtual than physical memory. With our custom policy, performance degrades more gracefully, but eventually if the matrix size is much larger than will fit in virtual memory, even our custom policy will perform poorly. 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). From the plot, we see that the number of page faults for the cus-

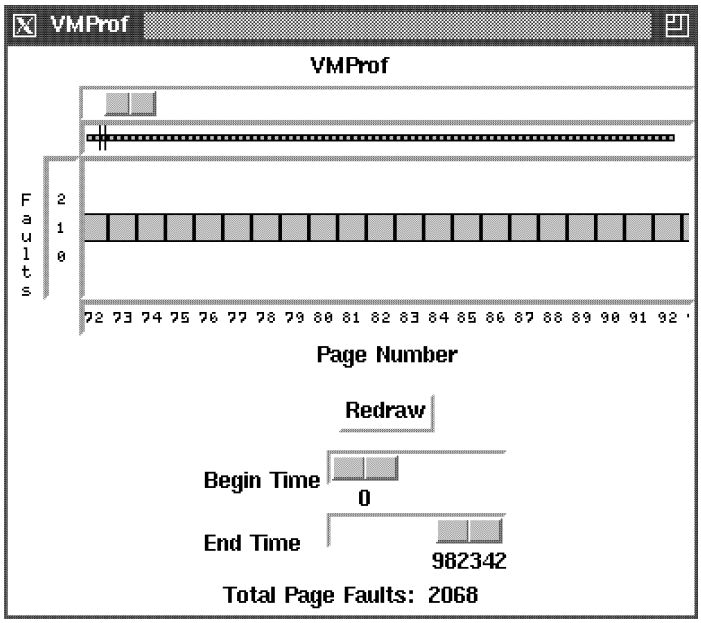


Figure 5: SOR with a Customized Page Replacement Policy

tom 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. 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 LRU.

Modifying the paging policy by itself does not help when the matrix is very large with respect to the amount of physical memory; instead, in order to have good performance in this case, we need to modify the application implementation to exhibit more spatial and temporal locality. Our original implementation scanned the entire matrix from beginning to end for 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, we can use a "blocked" implementation of SOR, where several relaxation steps are performed during a single sweep through the array. For instance, once we have computed the relaxation for rows 1 to $r$ during iteration $i$, we 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 we choose $r$ to be smaller than the size of physical memory, we can compute more relaxation steps for the same number of page faults relative to the original implementation.

Even the blocked version of SOR can benefit 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 can further improve the performance of the SOR application. Pre-fetching 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 ap-

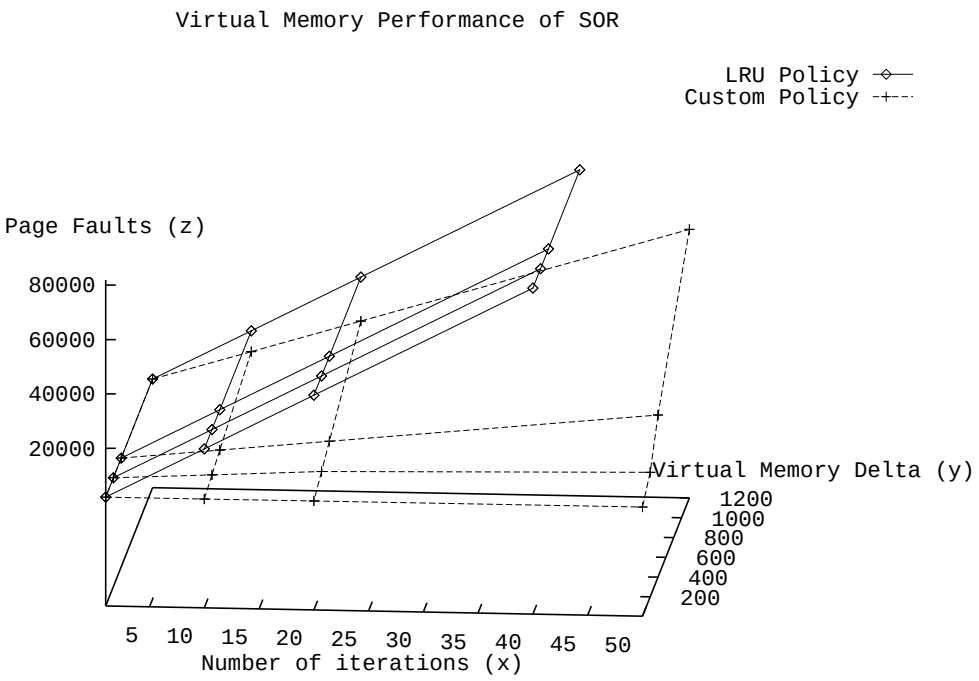


Figure 6: Number of Page Faults with SOR

plication 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.

# 7 Discussion

Application-specific virtual memory is an instance of a larger trend towards structuring system software to allow application control over policy decisions. Other examples include thread scheduling [Anderson et al. 1992], interprocess communication [Bershad et al. 1991], compiler optimizations [Steele Jr. 1990], database access routines [DeWitt & Carey 1984, Stonebraker 1987], and desktop publishing [Ald 1992, Clark 1992, Dyson 1992]. In all of these cases, an application-specific structure 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. In our view, a key ingredient to exploiting the potential of this general approach is to provide tools to simplify the job of developing application-specific system software.

One way of viewing this trend is in terms of meta protocols [Kiczales et al. 1991, Kiczales et al.

1992, Vahdat 1993]. Using virtual memory as an example, there is a very simple *base* protocol: reads and writes of the process's virtual address space. As we have argued, the choice of implementation of this abstraction (the virtual memory system) sometimes has a large impact on the performance seen by the user of the abstraction (the application). One approach is to leave the original interface alone, and instead to define a *meta* protocol, by which the user of the interface can select an implementation more suited to its needs.

Prior to our work, a number of systems already provided a meta protocol for virtual memory. Mach, V++, and Apertos all allow applications to select their own virtual memory managers, with no change to application code. Our work is aimed at making it easy for application programmers to exploit this flexibility, essentially by defining a *higher level* meta protocol. Application programmers still have the flexibility to completely re-write the virtual memory system, but usually it will be the case that they only need to modify a few lines of code of our extensible virtual memory system to adjust the paging policy. In this, we are essentially providing a *meta object* protocol, a meta protocol built using object-oriented techniques.

Prior work has identified four desirable characteristics for meta protocols [Kiczales et al. 1991]:

- *Incrementality* — making a small change to an implementation should require the user of the implementation to write only a small amount of code [Kiczales & Lamping 1992]. In the virtual memory domain, changing from an LRU to an MRU policy should only require writing new policy code – the actual mechanism for swapping pages in and out of main memory should be able to be reused.

- *Scope Control* — the effect of policy changes should be local to the application making the changes. This is automatically the case with application-specific virtual memory. Ideally, the scope of changes can be further restricted to applying to only the relevant parts of the application.

- *Interoperability* — if an application chooses to use its own implementation of an interface, it should be able to freely interact with other applications (or other parts of the same application) using the original implementation.

- *Robustness* — the behavior of the system should be graceful in the face of bugs in the application-specific code.

In our work, incrementality and interoperability are provided by our object-oriented extensible design. Scope control and robustness result simply from moving virtual memory management policy to the user level.

It is also important to note the limitations associated with meta protocols: one cannot always turn to the meta protocol to modify implementation decisions in the face of inadequate performance. With virtual memory, there are instances when lack of available memory causes very poor performance of an application 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.

In work related to our own are several research efforts in the operating system community which 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.

## 8 Conclusions

In conclusion, we have described an extensible user-level virtual memory system and a graphical tool to help programmers tune virtual memory performance. Together, these allow users to easily experiment with various page replacement policies and to get immediate feedback from the user interface. This feedback may be used to develop a paging policy that better meets the application's demands, or sometimes to modify the application to exhibit better paging behavior. Our structured virtual memory implementation allows users to easily modify a complex system, and the user interface provides a facility to evaluate the different tradeoffs involved with modifying operating system policy.

## 9 Acknowledgements

We wish to thank David Cheriton, Doug Ghormley, Kieran Harty, Paul Hilfinger, Gregor Kiczales, John Lamping, Luis Rodriguez, and the referees for providing many helpful comments on drafts of this paper.

## References

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

[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*, 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 et al. 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 Multiprocessor Cache Behavior: A First Experience. In *Shared Memory Multiprocessor Symposium*, Tokyo, Japan, April 1991.

[Clark 1992] Clark, J. *Window 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 *Proc. 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 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*, pp. 187–197, 1992.

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

[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*, pp. 435–451, 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 Accommodate User-Level Page Replacement Policies. In *Proceedings of First USENIX Mach Symposium*, Burlington, Vermont, October 1990.

[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.

[Vahdat 1993] Vahdat, A. The Design 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 *Proceedings of the 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.