SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(5), 547–571 (MAY 1997)

Hipec: A System for Application-customized Virtual-memory Caching Management

PAUL C. H. LEE, RUEI-CHUAN CHANG
Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan 30050, ROC

AND

MENG CHANG CHEN
Institute of Information Science, Academia Sinica, Taipei, ROC

SUMMARY

Conventional operating systems employ a kernel-controlled caching strategy that cannot properly serve all access-pattern types used by applications. When running under these systems, many memory-intensive applications with mis-matching access patterns cause excessive page faults and page replacements that reduce the application's performance. This paper presents the hipec system, which allows applications to have their own caching strategies for managing page frames with negligible overhead. Since application designers know the access patterns of their applications, the specific caching strategies can be tuned to meet the needs of each application. Empirical results show that the hipec system significantly improves application performance and system throughput. ©1997 John Wiley & Sons, Ltd.

KEY WORDS: *hipec*; virtual memory; caching; interpretation; IPC; upcall

INTRODUCTION

Conventional virtual-memory management schemes provide each application with a large address space by sharing a page frame pool among all applications. System kernels manage the page frame pool to cache each application's virtual memory. As the kernels do not know the applications' access patterns, the caching decisions of kernels cannot properly serve all application needs. Many memory-intensive applications run poorly under conventional systems when decisions taken do not meet their access-pattern requirements.[1,2,3,4]

Since application designers know their applications' access patterns, their knowledge can be used to make intelligent cache management decisions. One approach is to implement a collection of popular caching strategies in the kernel. Applications could then inform the kernel of their caching strategies by selecting one strategy from that collection. Since the strategies supported are finite, some applications may not be able to take advantage of this approach, because their preferred strategies were not implemented in the collection. Lack of flexibility is thus the major drawback of this approach.

Another alternative is to partition the management of page frames to the kernel and user applications. The kernel only handles the allocation of page frames, while the applications are responsible for making page-replacement decisions. When free page frames are exhausted, the

CCC 0038–0644/97/050547–25 $17·50
©1997 by John Wiley & Sons, Ltd.
Received 25 August 1995
Revised 4 September 1996

kernel selects victim applications and asks them to surrender page frames. User applications can then decide which page frames to give up in line with their specific access patterns. Previous systems[5,6,7] generally employed domain-crossing approaches to request the user-level replacement decisions. Unfortunately, domain-crossing approaches usually add significant complexity to the kernel, and create significant performance overhead for applications.

Problems of domain-crossing

Existing systems that employ the domain-crossing approaches generally implement asynchronous communication schemes to delegate the caching decisions to applications.[6,7,8,9] The purpose is to prevent the kernel from synchronously waiting for user-level managers' respondence. The implementation, however, added significant complexity to the kernel. The increased complexity comes from the extra flags added to the kernel-maintained data, the lengthy processing routines and the scattered checking statements. If a system wants to delegate applications, not only the page-replacement decisions but also other finer-grained caching decisions, the kernel complexity will be further increased.

Performance overhead is another problem for the domain-crossing approaches. To cross domains needs to do a context switch, which needs to flush the TLB, the cache memory, to allocate user-level stacks[10] and to update many kernel-maintained data, such as the *task*, *thread*, the virtual memory management data and the kernel stack. The impact due to flushing the TLB and cache memory is not small and cannot be ignored.[11,12] Moreover, the user-level decision-making managers are subjected to scheduling delay. Other applications compete with the managers for the CPU when the managers want to make their caching decisions. This delay is unbounded, and depends only upon when the managers are scheduled to run.

User applications generally need kernel-maintained data in making their specific caching decisions. The information needed includes, for instance, the referenced and modified bits of each page frame and the caching status indicating which regions of virtual memory have been cached. On the one hand, the kernel needs to provide an interface so that applications can invoke the interface to access the necessary information. The invocations, however, increase the incidences of domain-crossing and scheduling delay. On the other hand, applications have to maintain user-level data for keeping information; thus, they can make caching decisions accordingly. To maintain the user-level data is another overhead cost for the applications.

Motivation

The operations to cross domain between the kernel and applications only cost hundreds of microseconds. The domain-crossing overhead, however, is far larger than the time taken to perform domain-crossing operations. Moreover, the degradation of application performance usually causes applications to occupy page frames for a longer time. The longer occupation time will incur higher page frame competition and increase the opportunities of page replacements, which usually incurs more disk I/O operations. From the reported values of previous systems, the incurred overhead is about 10–15% to the application's performance.[6,7]

The original purpose of delegating caching decisions to applications is to incorporate the applications' knowledge into the memory cache management. Based on this, whether or not to perform domain-crossing operations is not important in achieving such a goal. If there is a mechanism whereby the kernel can be informed of the applications' knowledge without performing domain-crossing operations, the impacts due to domain-crossing overhead can be minimized.

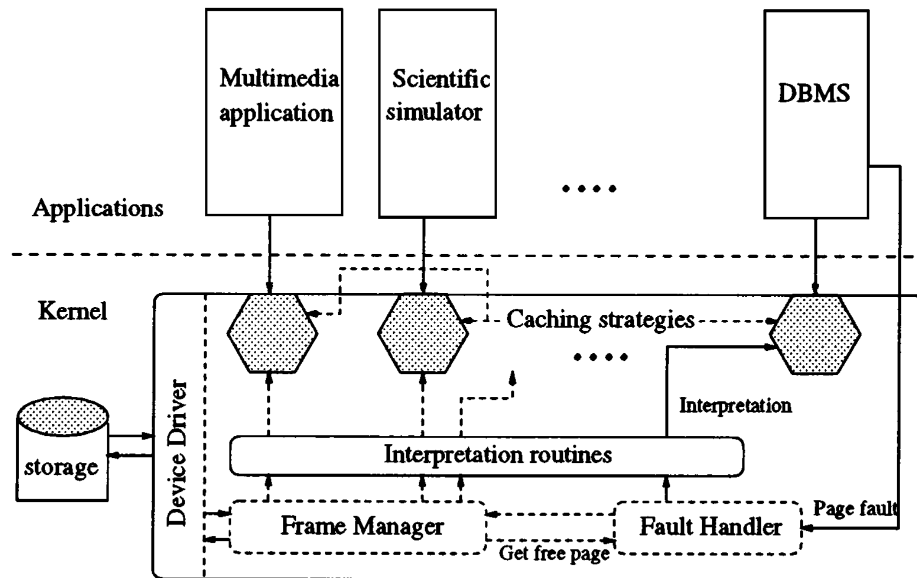

Figure 1. Overview of hipec system

This paper presents a **hi**gh **p**erformance **e**xternal virtual-memory **c**aching (**hipec**) system that supports application-specific virtual-memory cache management without domain-crossing overhead. Under the *hipec* system, each specific caching strategy is represented by a sequence of macro-like commands, called *hipec* commands. By loading the strategies into the kernel address space, the kernel is informed of the applications' caching decisions. When needed, the kernel synchronously interprets the caching strategies and performs the corresponding page-frame management operations for applications. The time needed to interpret a command is hundreds of nanoseconds, which can almost be ignored.

SYSTEM OVERVIEW

The *hipec* system is implemented on the Mach-based[8] OSF/1 MK5 operating system and uses the Mach External Memory Management (EMM) interface to perform paging operations on the memory-mapped regions. Caching operations on each virtual-memory region are performed by the *hipec* system, but under the direction of each application. The core of the *hipec* system is its representation of application-specific caching strategies, and the interpretation of the strategies to obtain the caching decision from each application. Rather than using a domain-crossing approach, *hipec* employs an *in-kernel strategy interpretation* approach to perform application-controlled virtual-memory cache management.

Figure 1 illustrates the design of in-kernel strategy interpretation. The application-specific caching strategies are loaded into the kernel when applications initialize the caching management of each virtual-memory region. When a page fault occurs, the page-fault handler allocates a free page frame to the faulting application, as in the DBMS shown in Figure 1, and calls interpretation routines to interpret the application's caching strategy. The DBMS uses that handling strategy to inform the page-fault handler of the designated actions to be

performed on the allocated page frames. For example, if this page frame is used to cache referenced-once data, the DBMS would tend to assign it a low priority, because this page frame is less important than frequently referenced pages.

When the number of free page frames is below a certain threshold, the frame manager starts reclaiming pages from applications. The frame manager selects victim applications and interprets their caching strategies to reclaim less important page frames. For instance, when the DBMS is selected to return page frames, it would tend to return the referenced-once pages on the low-priority list because those page frames are referenced only once, and will not be referenced again. The page-frame assignment and reclamation decisions of the DBMS are directed by its specific caching strategy. The page-fault handler and frame manager interpret that strategy and perform the designated operations for the DBMS.

Privileged and unprivileged applications

If applications are allocated sufficient page frames to meet their working set size, and the allocated page frames are not to be reclaimed during execution, the applications can run with a minimum numbers of page faults. As some applications are more critical than others, applications are designated as *privileged* and *unprivileged* under the *hipec* system. In *hipec*, the requested page frames are allocated to privileged applications upon initialization of the *hipec* services, and are free from reclamation during application execution. Allocation requests from a privileged application will be rejected only when the remaining free page frames are insufficient and all other page frames are possessed by privileged applications. Only privileged users are admitted to execute the privileged applications. By contrast, unprivileged *hipec* applications must share page frames with other unprivileged applications.

Other components

In addition to the in-kernel interpretation scheme, *hipec* employs a *FIFO*2-*MR* (**F**irst **I**n, **F**irst **O**ut, with Second-chance, **M**ovement and **R**ecovery) page-frame reclamation policy to globally share page frames among applications. *Hipec* also includes a detection mechanism to protect the system from any ill-programmed caching strategy, and auxiliary tools to help application designers in designing their specific caching strategies.

CACHING STRATEGY REPRESENTATION

Each specific caching strategy is represented as a sequence of *hipec* commands. Each command represents a basic virtual-memory management operation for managing page frames. *Hipec* introduces a new kernel object, the *hipec* container, in which to store *hipec*-related information. The *hipec* commands, container and the caching-strategy structure are introduced in following sections.

Hipec commands

The *hipec* commands are a set of 32-bit commands, composed of an 8-bit operator code and up to two operands. A major concern in the design of *hipec* commands is deciding how many and what kinds of commands are sufficient for applications to implement their caching strategies. The command set should be at least as flexible as the kernel in that various caching

Table I. The *hipec* command set

No.	Command	Binary	Operations
1.	Return	00000000	The end of execution.
2.	Arith	00000001	Arithmetic operation.
3.	Comp	00000010	Comparison operation.
4.	Logic	00000011	Logical operation.
5.	EmptQ	00000100	Test whether a specified queue is empty.
6.	InQ	00000101	Test whether a specified page is on the specified queue.
7.	Jump	00000110	Branch to the next command.
8.	Dequeue	00000111	Move a page from a queue.
9.	EnQueue	00001000	Add a page to a specified queue.
10.	Request	00001001	Request page frames from the system.
11.	Release	00001010	Release page frames to the system.
12.	Flush	00001011	Flush a page frame.
13.	Set	00001100	Set or reset the referenced or modified bits.
14.	Ref	00001101	Test whether a specified page has been referenced.
15.	Mod	00001110	Test whether a specified page has been modified.
16.	FindVA	00001111	Find the virtual address of a specified page.
17.	FindPA	00010000	Find the physical page when given its virtual address.
18.	Map	00010001	Map a page to a specified virtual address.
19.	UnMap	00010010	Unmap a specified virtual page.
20.	Call	00010011	Invoke another policy event.

strategies can be implemented in the command set without limitations. A coarse-grained interface, such as that exported by Reference 14, is sufficient to support most applications. However, it is not sufficient to support sophisticated caching strategies, such as the strategies for assigning different priorities to different segments of a file. On the other hand, since *hipec* is designed to support user-controlled virtual-memory cache management, it is not necessary to supply a fine-grained interface to operate the hardware architecture and the operating system internals. The *hipec* command set, as listed in Table I, should be flexible enough to support most virtual-memory caching strategies.

Hipec **container**

The most important mission of the *hipec* container is to act as an operand pool. As *hipec* commands are defined to support basic page-frame management operations, the data types of the command operands are limited to *Integer* and pointers to *Integer*, *Page* and *Queue*. The *Page* is the data structure that the Mach kernel uses to manage physical memory, and the *Queue* is a data structure used as the header node of any page-frame list. The operand variables are stored in the *operand array* of the *hipec* container. Each operand field of *hipec* commands stores an index to the operand array to indicate the accessed operands. When applications invoke the *hipec* service, a caching strategy segment, the *Init* event, is interpreted by the kernel to initialize the operand array. The initialization fills in the initial values of integer operands and creates queue headers for the *pointer-to-Queue* operands. The relationship among the

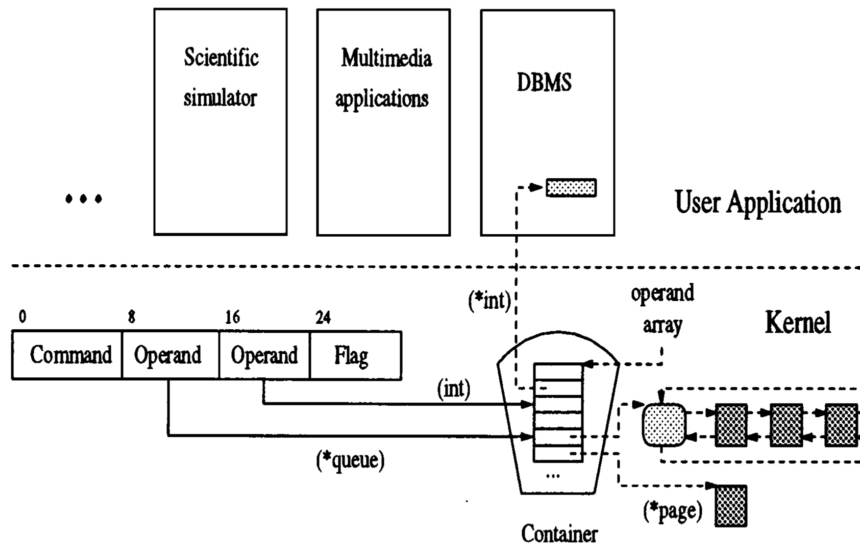

Figure 2. The hipec container

hipec commands, the *hipec* container and the operands is shown in Figure 2.

Application-specific caching strategies

Each specific caching strategy contains at least three *hipec* command segments: the *Init* event, the *PageFault* event and the *Replace* event. When the *hipec* service is initialized, the Init event is interpreted by the kernel to initialize the operand variables, as described in the previous section. When a page fault occurs, the pager-fault handler maps a page frame to the faulted address, and invokes interpretation routines to interpret the PageFault event of the faulting application. User applications use the PageFault event to instruct the page-fault handler to perform necessary management operations on the allocated page frames, such as recording the physical and virtual addresses of mapped page frames or assigning priority to them.

The Replace event is interpreted under different conditions for privileged and unprivileged applications. A privileged application's Replace events are interpreted when the allocated number of free page frames has been exhausted. Privileged applications use the Replace event to request more page frames from the system, or to do page replacements from their private page-frame pools. An unprivileged application's Replace events are interpreted when the number of free page frames of the system is low. The frame manager selects victim applications and starts reclaiming page frames from them by interpreting their Replace events. The difference comes from the fact that privileged applications are allocated private page-frame pools when the *hipec* service is initialized, while unprivileged applications must share the page-frame pool with other unprivileged applications. Such applications use the Replace event to select the least important pages for replacement. Figure 3 gives an example of a specific caching strategy.

hipec Auxiliary Tool

hipecGen | hipecProf | Quit

hipec Generator

Quit | Files | Edit | Translate to

```
Event Init(){
    *Queue      active_queue;   /* The _free_queue and _free_count*/
    Int active_count;           /* are default assigned to the    */
    *Queue      inactive_queue; /* operand[0] and operand[1].     */
    Int inactive_count;         /* Application does not to declar */
    Int reserver_free_target=1; /* them.                          */
    Int free_target;
    Int inactive_target;
    *Page       page;
}
Event PageFault()
{
    page = FindPA(FaultedAddress);
    _free_count--;
    EnQueue(page, active_queue, TAIL);
    active_count++;
}
Event Replace()
{
    For(;;){
        inactive_target = (active_target + inactive_target) * 2/3;
        while(inactive_count < inactive_target)
        {
                page = DeQueue(active_queue, HEAD);
                active_count--;
                Set(page, REFERENCE, RESET);
                EnQueue(page, inactive_queue, TAIL);
                inactive_count++;
        }
        if(_free_count < free_target){
                page=DeQueue(inactive_queue, HEAD);
                inactive_count--;
                if(Ref(page))
                {
                        EnQueue(page, active_queue, TAIL);
                        active_count++;
                }else{
                        if(Mod(page))
                                Flush(page);
                        EnQueue(page, _free_queue, HEAD);
                        _free_count++;
                }
        }else
                break;
    }
}
```

```
unsigned event[0]={ MAGIC,
02 02 00 00,
03 01 00 00,
00 00 00 00,
04 02 00 00,
05 01 00 00,
00 00 00 00,
06 01 00 00,
00 00 00 00,
07 01 00 00,
00 00 00 08,
08 01 00 00,
00 00 00 00,
09 03 00 00,
0A 01 00 00,
00 00 00 00,
0B 01 00 00,
00 00 00 02,
0C 01 00 00,
00 00 00 03};/*end*/
unsigned event[1]={ MAGIC,
02 01 06 04,
06 00 00 04,
10 03 00 00,
07 09 00 01,
01 01 00 06,
08 09 02 02,
01 03 00 05,
00 09 03 01};/*end*/
unsigned event[2]={ MAGIC,
01 0A 03 01,
01 0A 05 01,
01 0A 0B 03,
01 0A 0C 04,
01 08 0A 07,
02 05 08 02,
06 00 00 0E,
07 09 02 01,
01 03 00 06,
0c 09 02 01,
08 09 04 02,
01 05 00 05,
06 00 00 06,
02 01 07 02,
06 00 00 1C,
07 09 04 01,
01 05 00 06,
0D 09 00 00,
06 00 00 17,
08 09 02 02,
```

Figure 3. Example of implementing the privileged First-In, First-Out with Second-chance caching strategy

```
vm_pageout_scan(){
        while(vm_page_free_count < vm_page_free_target) {
               page = dequeue(vm_page_inactive_queue);
               if(page->referenced | |   (page->hipec == PRIVILEGED) | |
                  pmap_is_referenced (page->phys_addr)) {
                               enqueue (vm_page_active_queue, page);
               } else {
                      if (page->hipec == UNPRIVILEGED) {
                            set_timeout(timeout_detecting_function, TimeQuantiumOfScheduling);
                            reclaim_page = interpreter(current_task->map->object->container->Replace);
                            reset_timeout(timeout_detecting_function);
                            if(reclaim_page != page) {
                                   location_movement(*page, *reclaim_page);
                                   insert_recovery(page, reclaim_page);
                                   page = reclaim_page;
                            }
                      }
                      if (page->dirty) flush(page);
                      enqueue_(vm_page_free_queue, page);
               }
        }
        while (vm_page_inactive_count < vm_page_inactive_target) {
               page = dequeue(vm_page_active_queue);
               pmap_clear_reference(page->phys_addr);
               page->referenced = FALSE;
               enqueue(vm_page_inactive_queue);
        }
}
```

Figure 4. FIFO2-MR page-frame reclamation policy

PAGE FRAME RECLAMATION

The page-frame management model of the *hipec* system leaves allocation responsibility to the kernel and delegates caching decisions to user applications. Obviously, different page-frame reclamation policies have different impacts on application performance as well as whole-system throughput. *Hipec* aims to provide unprivileged applications with a fair share of page frames by employing the FIFO2-MR policy, detailed in Figure 4. The FIFO2-MR policy was adapted from the Mach FIFO2 policy by adding the fundamental concept of the LRU-SP file-system allocation strategy.[13,14] The LRU-SP was selected because this policy maintains the fairness of page-frame allocation and prevents applications from having to suffer because of other applications' ill-advised caching decisions. The fundamentals, evaluations and implementations of LRU-SP are described in References 13 and 14.

Implementation

The Mach kernel manages page frames in the *active*, *inactive* and *free* queues, and employs the FIFO2 page replacement policy[15] to reclaim page frames. The *hipec* system also maintains the *free*, *active* and *inactive* queues to manage the page-frame pool, and employs the FIFO2 policy to select victim page frames. Unlike Mach, the *hipec* frame manager does not reclaim

victim page frames directly. If the selected victim page frame belongs to a privileged application, the frame manager will select the next page frame as a victim page and proceed with checking. If the victim page frame belongs to an unprivileged *hipec* application, the frame manager will interpret the Replace event of that victim application and reclaim the page frame specified by the Replace event.

The concepts of *Swap* and *Place Holder* in the LRU-SP policy have been imported into the FIFO2-MR. Before reclaiming application-suggested page frames, *hipec* moves the frame-manager-selected page from the head of the inactive queue to the location of the application-suggested page. This operation is called *location movement*, because only one page frame is moved. The application-suggested page is then reclaimed and placed in the *free queue*. Without the location movement, the same page frame would be repeatedly selected as a victim page, thus causing the same application to be repeatedly selected as the victim application.

An application-specific caching strategy is *ill-advised* if the replaced page is referenced before the original frame-manager-selected victim page. An ill-advised caching strategy will cause the application to create repeated page faults, and to be allocated excessive page frames due to the repeated faults. The excessive allocation will result in competition for page frames among applications, and cause other applications to suffer from excessive page replacements. To ease the impact of ill-advised strategies, a *recovery hash table*, motivated by the LRU-SP Place Holder, helps in recovery from ill-advised caching decisions. The recovery hash table is a 64-entry array that is created for each virtual-memory region when the caching management service of that region is initialized. Each entry in the hash table points to a list of *recovery* data structures that record user-controlled page reclamation decisions, as indicated in Figure 5. When the frame manager finds that an application-suggested page frame is different from a frame manager-selected victim page, the offsets to the virtual-memory region of the two page frames are recorded in the recovery data structure. When a page fault occurs, the page-fault handler hashes into the recovery hash table to check whether a recovery data structure was built for the faulty address. If one exists, and the referenced bit of the *select_but_resident* page is not set, the page-fault handler will reverse the ill-advised page replacement decision by reclaiming the *select_but_resident* page, mapping its physical page frame to the faulted address and clearing the recovery data structure. This recovery operation prevents applications with ill-advised caching strategies from monopolizing system page-frame resources and protects applications from being penalized by the ill-advised page replacement decisions of other applications.

PROTECTION MECHANISM

Hipec protects system resources from direct accesses by applications. User applications can only inform the kernel of their caching decisions using *hipec* commands. Page-frame management operations are performed by the kernel, and applications cannot directly modify kernel resources. However, it may still be necessary to protect the system further because of possible misbehaved caching strategies. The potential safety problems and the protection mechanisms are discussed as follows.

Dangling references in *hipec* commands

As with any other programming language, the operand fields of *hipec* commands may refer to non-existent variables. If it receives erroneous commands, the system will refer to wrong operands or destroy other kernel data. To avoid dangling references, *hipec* checks the syntax

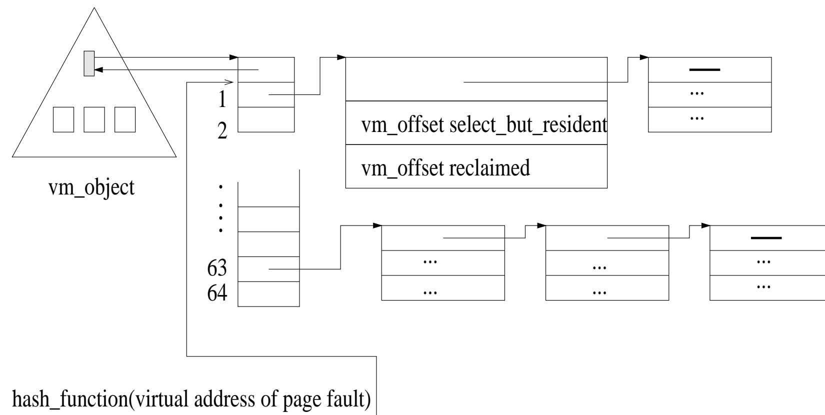

Figure 5. Recovery data structure

of the commands when loading any caching strategy into the kernel. As each operand variable is initialized prior to loading the caching strategy, the *hipec* system can check the existence and data type of every operand.

Infinite strategy interpretation

Another safety problem the *hipec* system deals with is long or infinite execution time for interpreting application-specific caching strategies. Since the interpretation is executed in kernel mode by the page-fault handler or by the frame manager, the interpretation operations cannot be preempted unless the interpretation is finished. The system will be monopolized by interpretation operations if a caching strategy interpretation forms an infinite loop. *Hipec* implements a detection mechanism to avoid infinite strategy interpretations.

When a page fault happens, the page-fault handler interprets the PageFault event of the faulting application. Before starting the interpretation, a detection function is initiated in the *callout* table.[16] When the registered duration expires, an interrupt is generated to interrupt the strategy interpretation and sets a *overtime* flag in the *hipec* container. Because the interpretation routine checks the *overtime* flag before interpreting a command, the interpreting routine will notice the expiration and invoke blocking routines to relinquish the processor. A command counter indicating the command currently being interrupted is saved into the container. The blocked application will be re-directed to a correct location in the process-run queue by examining its system resource consumption, and be scheduled to continue the strategy interpretation. This detection mechanism repeats as long as the interpretation continues. Only the application itself suffers from its infinite strategy interpretation.

When the frame manager invokes the interpretation routine to interpret a Replace event from an unprivileged application, the *overtime*-detection mechanism is similar to that of the

page-fault handler. The difference is that, when the interpretation of a Replace event is not finished within the legal duration, the frame manager will give up the interpretation, reclaim the original-selected victim page directly, and record the reclamation information in the *hipec* container to inform the application about the reclamation. The time spent in interpreting the Replace event is counted when calculating the scheduling priority for each victim application.

AUXILIARY TOOLS

Hipec supplies two auxiliary tools to help application designers prepare their specific caching strategies. The first is the *hipec* profiler for identifying performance problems with particular applications. The second is the pseudo-code translator, used to translate pseudo-code programs into *hipec* commands. Both tools are integrated into a single development environment. Application designers can directly write or modify their caching strategies, translate them into *hipec* commands and run the *hipec* profiler. After identifying any performance problems with the profiler, the application designers can then use the pseudo-code editing panel to modify their caching strategies.

Hipec profiler

The *hipec* profiler is a visual tool to help application designers view the caching behavior of their particular memory caching strategies. The inputs to the *hipec* profiler are the application trace files and the specific caching strategies in *hipec* commands. The output is a graphic display that shows the number of page faults in each virtual-memory region. Given the starting and ending time, the profiler graphically shows the page-fault numbers for that time window. Application designers can use the scroll bar to isolate a region of virtual address space and identify regions with high page-fault rates. Unlike a previous related implementation,[5] the *hipec* profiler simulates multiple applications simultaneously. Both the process scheduling policy and global page-frame allocation policy can be customized to experiment with interactions between simulated applications and the system. By selecting any application of interest from a scrolled list, users can observe its paging behavior in the bottom portion of the window, as illustrated in Figure 6. The top portion of the window in Figure 6 displays the paging behaviors of all the applications.

The *hipec* profiler also dynamically shows the page frame allocations of all the applications. In Figure 7, each bar represents a simulated virtual address space, and the shadowed area shows the regions cached by page frames. By specifying the starting and ending time, page frame allocations and competition among applications are displayed for the assigned time window. This tool helps application designers and system administrators view interactions among competing applications, and modify the caching strategies, scheduling policies and the global page-frame allocation policies to increase system throughput. A case study is described in the following paragraph.

Three trace files are selected from our experimental benchmarks. They are a Random Data Retrieval (RDR) operator that uses a tree-structure index, a Nested-Loop Join (NLJ) operator and a sequential MPEG (SMPEG) player . The detailed descriptions of the behaviors of the benchmarks are described in the section on performance evaluations. The RDR operator runs with a 16 MB index file and a 64 MB of data. The NLJ operator has 16 MB of outer relations and 2 KB of inter relations. The SMPEG player plays a 4.17 MB MPEG video file. The global page-frame allocation policies include the FIFO2-MR and modified Automatic Working Set Trimming (AWST) policies. The AWST policy was adapted from the replacement policy used

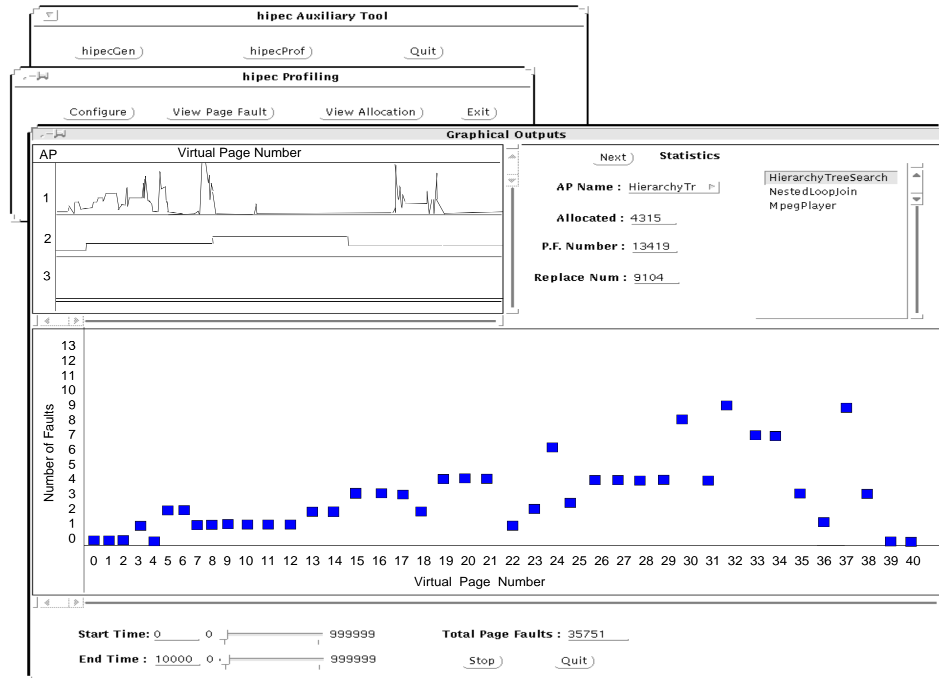

Figure 6. Graphical display of page faults

by Windows NT.[17] The profiler runs with the same round-robin scheduling policy but with different job-submit orders for the simulated benchmarks. The simulated results are shown in Figure 8.

The modified AWST requires two parameters to allocate page frames to applications, a *min* allocation and a *max* allocation. The modified AWST employs the FIFO page replacement policy to reclaim page frames, but each application is guaranteed an allocation of at least the *min* number of page frames, and can get at most the *max* number of pages during execution. The graphic output of page frame allocation shows the NLJ operator to be sensitive to the *max* value and to competition from other applications. The other two benchmarks are not so sensitive to the *max* value if it is not too small. Based on this information, the AWST can be modified to support different min/max values adapted to different applications. By extending the *max* value of the NLJ operator to 16 MB and restricting the *max* value of the other two applications to 1 MB each, the system will minimize the number of page faults, as is shown in Figure 8, which depicts the AWST(512K, x M) policy. Though the dynamic AWST performed better than FIFO2-MR in our simulation, this policy is not implemented in the *hipec* system. The dynamic AWST is not fair in allocating page frames to applications. In addition, it is hard to determine the right min/max values for applications.

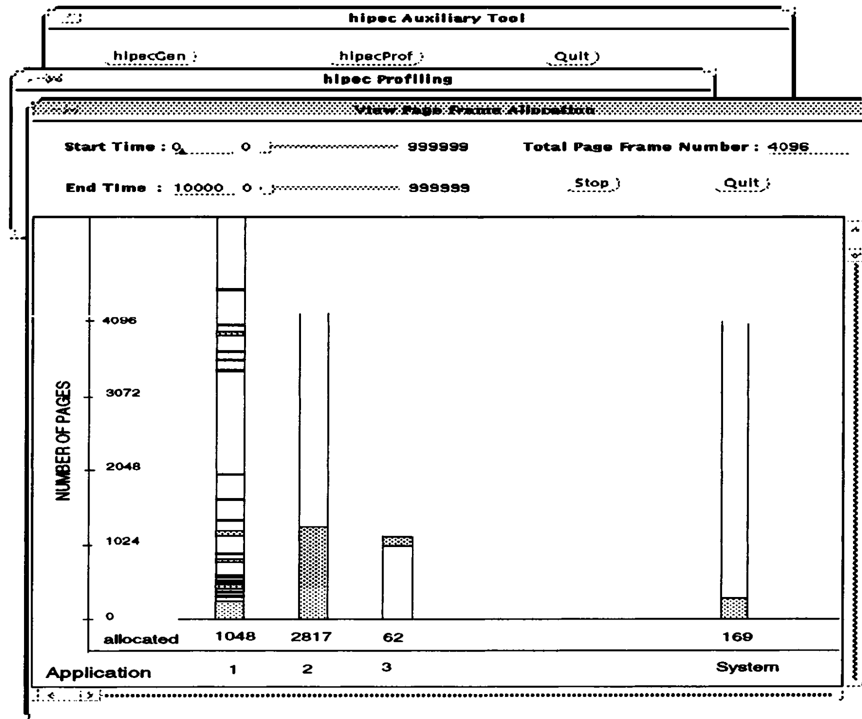

Figure 7. Graphical display of page-frame allocations

Pseudo-code translator

A Pseudo-Code Translator (PCT) is included in *hipec* to help application designers design their specific caching strategies in C-like programming language. PCT translates each segment of pseudo code into a sequence of *hipec* commands and can feed them directly into the profiler. Although the exported operations, data types and language constructs are limited to *hipec* commands, the PCT eases the programming of specific strategies for allocating operands and assigning corresponding operand numbers.

PERFORMANCE EVALUATIONS

Several aspects of the *hipec* system were evaluated. First, the page-fault processing time and the elapsed time of an experimental benchmark were used to evaluate *hipec* overhead. The second experiment gave the time to perform a domain-crossing operation, to interpret a command and the disk transfer time. Following that experiment, *hipec* was further investigated by evaluating the RDR operator. The performance improvements in RDR resulting from applying the strategy interpretation and domain-crossing approaches were compared. In addition, the overhead created by the recovery scheme was evaluated. Following this series of overhead analyses, the next experiment evaluated three privileged *hipec* applications. These applications have common access patterns,[18,19] and showed great performance improvements when intelligent

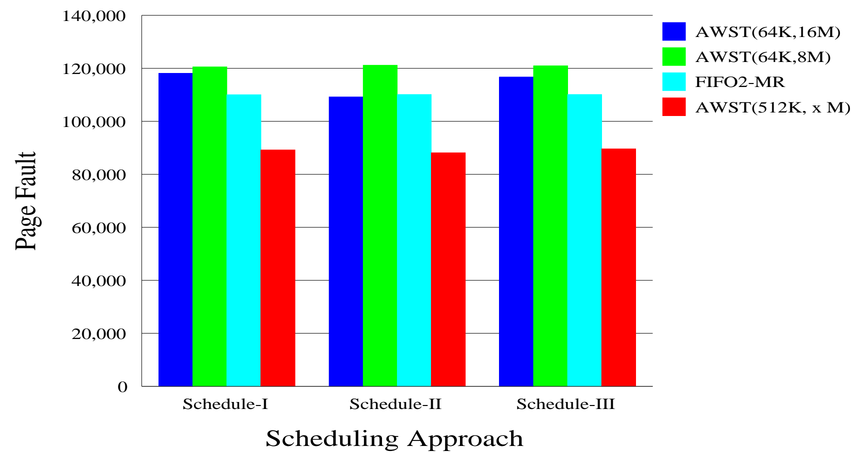

Figure 8. Simulation results under FIFO2-MR and AWST policies

cache management was added. The *hipec* system throughput was also compared to that of the Mach with these applications running concurrently. Finally, the recovery scheme was proven to significantly protect applications from ill-advised strategies in the last experiment. An Acer 486-33 PC was used as our experimental platform.

System overhead

The *hipec* overhead includes the time required to interpret *hipec* commands, to do the necessary checking, and the time required for performing recovery operations and detecting infinite strategy interpretation. An experimental benchmark that implements the same Mach FIFO2 caching strategy was employed in this experiment, running as an unprivileged *hipec* application. The page-fault processing time of *hipec* and Mach is listed in Table II. Note that the time listed does not include the time for doing disk I/O. As shown in Table II, only 2% of the page-fault processing time was increased, much less than the reported value in a previous study.[20]

This evaluation only highlights *hipec* system overhead in processing page faults. To further investigate *hipec* overhead in performing application-specific cache management, the elapsed time of the benchmark was evaluated. This benchmark sequentially wrote data to a 60 MB virtual-memory region that created page faults and dirty pages. When free page frames were exhausted, the dirty pages were replaced and flushed to disks, thus causing disk I/O operations. Since the free page frames are about 58 MB in size, sequentially writing data to a 60 MB region definitely causes page replacements. Table II shows that less then 2% of the page-fault processing time was created by the *hipec* system. Though the listed value cannot be used to predict the *hipec* overhead for all applications, the value can at least be used as an indicator to show that the *hipec* system creates only negligible overhead, which can be easily compensated for by eliminating a few disk I/O operations.

Table II. *hipec* overhead

Evaluations	Averaged elapsed time Overhead
Page-fault processing time of *hipec* system	400.60 μsec
Page-fault processing time of Mach	392.24 μsec
hipec-created overhead	2.06%
60 MB sequential access	
Running under original Mach kernel	83515.6 msec
Running under *hipec* system	85102.4 msec
hipec-created overhead	1.9%

System parameters

This evaluation gave a rough comparison of the domain-crossing approaches and the interpretation scheme. Table III lists the time required to interpret a *hipec* command and the time for performing a message-passing IPC and a system call. The system call is evaluated as a reference to Upcall. Upcall needs to allocate user stacks before transferring control from kernel to application,[10] which creates more overhead than the system call. As listed in Tables II and III, the time required to process a page fault is close to the time needed to perform a single IPC operation. This means that, if the kernel delegates the caching decisions to applications in processing a page fault, the cost to perform an IPC cannot be ignored. However, the time to interpret a *hipec* command is much shorter than required by any domain-crossing approach.

There might still be concerns about the benefit of employing the strategy-interpretation approach. If applications can intelligently manage page frames to earn huge performance gains from reducing the number of disk I/O operations, the domain-crossing overhead would be negligible. Based on this suspicion, the averaged disk I/O transfer time was evaluated for comparison with the time required for domain-crossing.

The disk I/O bandwidths were evaluated by sequentially reading 10 MB of data via the raw OSF/1 MK5 device interface. Two different SCSI disk devices were evaluated. One was a Segate 31230n disk plus an AHA-1742 adapter, the other was an IBM Pegasus disk and an AHA-2940 adapter. As shown in Table III, the averaged disk I/O transfer time for 4 KB of data is only about 3–5 times greater than the time used by the Mach IPC and 38–60 times greater than that required for system calls. Thus, applications with sophisticated caching strategies that require huge kernel/application interactions would incur significant domain-crossing overhead that cannot be ignored in evaluating application performance. This overhead can, of course, be reduced by applying the strategy-interpretation approach. The next experiment identified the performance improvements achieved by applying the strategy-interpretation approach.

Performance improvements yielded by strategy interpretation

In this experiment, the *hipec* system was modified in several ways to employ domain-crossing approaches for comparison with strategy interpretation. First, the functions of the basic *hipec* commands were implemented as system calls. Second, application-specific caching strategies were implemented as executable programs which would invoke the additional system calls for getting kernel-maintained information. Third, the *hipec* system used message-passing

Table III. Time for domain-crossing and disk I/O

Evaluations	Averaged Time
hipec Command Interpretation	smaller than 150 n sec
Single Null System Call	19.4 μsec
Single System Call with 32 byte arguments	24.8 μsec
Single Null IPC Call	292.3 μsec
Single IPC Call with 32 byte arguments	317.9 μsec
SCSI Disk I/O bandwidth (Segate 31230N+AHA1742)	2.6 MB/sec
Averaged transferred time (4 KB)	1502.4 μsec
SCSI Disk I/O bandwidth (IBM Pegasus+AHA2940)	4.1 MB/sec
Averaged transferred time (4 KB)	952.7 μsec

IPC to transfer control to applications. When notified by the kernel, the applications executed their caching strategies and informed the kernel of their caching decisions using system calls.

The Random Data Retrieval (RDR) operator is used to exemplify the performance difference. The RDR randomly retrieves data tuples via a 16 MB index file. The structure of the index file is a highly balanced binary tree. Each index record is 64 bytes long, and consists of a primary index key, and pointers to data tuples and left/right index records. The data file has 64 MB of data and each data tuple is 64 bytes long.

The RDR ran as an unprivileged application and used the *multi-queues*, *prioritized*, *FIFO*2 caching strategy (MPFIFO2), as shown in Figure 9, to manage page frames. Page frames that cache the index records from the same *index-tree level* have the same priority and are linked in the same queue. The *index-tree level* of the index record is defined as the length of the path from the root record to the index record by traversing the tree-structure index file. The queues for linking page frames with low *index-tree levels* have high priorities and will not be selected to return page frames if any non-empty lower-priority queues exist. Each prioritized queue is independently managed by a *FIFO2* caching strategy. The RDR was evaluated using combinations of different numbers of retrieved tuples and system page-frame sizes. As shown in Table IV, the rows *hipec* and *hipec+domain-crossing* represent, respectively, evaluations of the *hipec* and the modified *hipec* using domain-crossing techniques. Evaluation results were normalized using data from the row *original Mach*, which are the results obtained when the RDR ran under the Mach system.

The results shown in Table IV demonstrate that the intelligent MPFIFO2 caching strategy does reduce the number of block I/Os for the RDR when compared with the number required by the Mach FIFO2 policy. No matter whether a domain-crossing approach or strategy interpretation was used, the RDR induced almost the same number of block I/Os; the difference is smaller than 0.1%. The elapsed time for the RDR, however, differed when using strategy-interpretation instead of domain-crossing. The difference, ranging from 1–5%, was caused by the domain-crossing overhead. This overhead is significant to the RDR operator, since the RDR only benefited from 1.5–15% of elapsed-time reduction yielded by the *hipec* system. Particularly, the performance difference will be further increased if the EMM interface is replaced by the *hipec* commands. The current *hipec* system uses the Mach EMM interface to perform disk I/O operations. However, the EMM is implemented by asynchronous IPC which caused large domain-crossing overhead.

Table IV. Evaluations of the RDR operator

No. of accessed		Page-frame size					
		12 MB		16 MB		12 MB	16 MB
		Elapsed time (in sec/normalized)				No. of block I/O	
10^5	Original Mach	8916.42	1.0000	6008.17	1.0000	203,833	137,287
	hipec	8440.28	0.9466	5917.32	0.9849	171,278	124,088
	hipec+domain crossing	8713.42	0.9772	6140.95	1.0220	171,334	124,160
	hipec-recovery	8057.77	0.9034	5722.78	0.9525	171,133	123,921
	hipec+recovery(1024)	8082.75	0.9065	5738.23	0.9551	171,199	124,003
10^6	Original Mach	98114.17	1.0000	60313.42	1.0000	2,242,611	1,378,580
	hipec	84378.31	0.8600	51296.06	0.8505	1,712,216	1,040,931
	hipec+domain crossing	85457.44	0.8710	52167.70	0.8649	1,714,239	1,043,316
	hipec-recovery	80990.60	0.8255	49583.66	0.8221	1,710,121	1,038,928
	hipec+recovery(1024)	81217.37	0.8278	49653.08	0.8233	1,710,381	1,039,339

Recovery scheme overhead

Recovery scheme overhead is another important factor that would dominate the performance improvements achieved by the *hipec* system. Recovery overhead results from inserting recovery information into the recovery table, and checking for any ill-advised caching decisions by searching the link lists of the recovery hash table.

The *hipec-recovery* row in Table IV lists evaluation results obtained using the same RDR operator, but disabling the recovery scheme functions. By comparing these results with those in the *hipec* rows, there are almost 3–4% savings in elapsed time. We suspect that most recovery scheme overhead comes from recovery-checking performed when searching recovery-link lists. When an application manages a large virtual address space, many recovery nodes will be created that cannot be hashed to only 64 entries without causing hashing conflicts. As a result, long-link lists are created to queue the recovery information.

Based on this suspicion, the recovery hash table was extended to 1024 entries to re-examine the recovery scheme overhead. The results thus obtained are presented in the row *hipec+recovery(1024)* of Table IV. The recovery overhead was greatly reduced compared to results obtained without extending the number of recovery entries. Extending the recovery hash table seems a good choice to reduce the recovery scheme overhead. The extension, however, is a trade-off between the memory size and elapsed time. The 1024-entry hash table is large enough for the RDR to access 10^5 and 10^6 tuples, but possibly insufficient for accessing more tuples.

Performance improvements of privileged applications

Privileged applications can own page frames during execution without competing with other applications. They also do not incur any recovery scheme overhead, and should run with maximum performance if granted enough page frames. In this experiment, the RDR operator, Nested-Loop Join (NLJ) operator and a MPEG (SMPEG) video player were used to explore performance improvements in privileged applications.

```
PageFault(){
            page =  FindPA ( FaultedAddress) ;
            if (  EmptyQ( IndexLevel )  );
                        CreateQ ( IndexLevel );
            Set ( page,   IndexLevel,   TAIL ) ;
            EnQueue ( page,  IndexLevel,  TAIL );
}
Replace(){
            Indicator = 25 ;
    NextLevel:
            if ( EmptyQ (Indicator) ) {
                        Indicator =   Indicator   -   1 ;
                        goto   NextLevel;
            } else do {
                        page = DeQueue ( Indicator,   HEAD );
                        if ( Ref (page) ) {
                        Set (page,   REFERENCE,   RESET) ;
                        EnQueue( Indicator,   TAIL ) ;
                  } else  if ( Mod(page) ) {
                              Flush (  page  ) ;
                        }
                        Release (  page  ) ;
                        return ;
                  }  while (TRUE) ;
}
```

Figure 9. The pseudo code of multi-queues, prioritized, FIFO2 caching strategy

The NLJ operator is one of the most important operations in relational DataBase Management Systems (DBMS). When running under the *hipec* system, the NLJ operator implements a MRU-like caching strategy. The inner relation of the NLJ operator is 4 KB long, and the size of the outer relation is 60 MB. Each tuple in both relations is 64 bytes long. The NLJ operator was evaluated using different number of allocated page frames.

The SMPEG player has a sequential access pattern that reads a 12.96 MB video data. SMPEG ran concurrently with a DataBase Generator (DBG) to show that system performance can benefit from privileged *hipec* service by eliminating page-frame competition. The video player uses a *FIFO* caching strategy with 16 page frames allocated. The DBG sequentially builds a 60 MB database, which is a general application that does not invoke the *hipec* service.

Because Mach consumes about 6 MB of physical memory after booting the system, the available memory left for applications is about 58, 42, 26 and 10 MB, respectively, when the Mach is booted with 64, 48, 32 and 16 MB of physical memory. To make the comparison fair, the applications were allocated 58, 42, 26 and 10 MB of physical memory, respectively, when running as privileged *hipec* applications.

By using the *hipec* system to reduce the number of page replacements and disk I/O operations, the performance improvements in each application are evident, as shown in Tables V, VI and VII. There was 16–95% of elapsed-time reduction for the NLJ operator and 4–13% for the concurrently run SMPEG player, and DBG generator. For the RDR operator, elapsed-time reductions ranged from 9–15% when the number of randomly retrieved tuples was 10^6.

Table V. Elapsed time of NLJ operator

	Allocated page frame size (MB)			
	64	48	32	16
	Elapsed time (in seconds)			
Mach LRU	12571.80	12749.97	12735.72	12779.17
hipec MRU	593.19	3817.52	7239.93	10694.29
Ratio	4.72%	29.94%	56.85%	83.69%

The RDR operator gained little performance improvement when the number of randomly retrieved tuples was small. Most of the accessed tuples could be cached by the allocated page frames, causing few page replacement operations. The intelligent caching strategy of the RDR operator helped a little under this condition.

The SMPEG accessed the video data in a sequential read-only pattern. Consequently, the SMPEG player could reuse its recently-accessed page frames so that 16 page frames were enough for it to play the video. Without privileged, application-specific cache management, the SMPEG would normally request free page frames from the system, causing more DBG dirty pages to be reclaimed. Reclaiming the dirty pages would require flushing them first, thus degrading system performance.

Table VI. Elapsed time of RDR operator

Access size		Allocated page frame size (MB)			
		64	48	32	16
		Elapsed time (in seconds)			
	Original	12.00	12.43	11.98	19.95
10^2	*hipec*	12.07	12.13	12.18	17.47
	Ratio	100.58%	97.59%	101.67%	87.57%
	Original	52.94	54.05	61.67	82.75
10^3	*hipec*	51.33	56.11	61.88	81.01
	Ratio	96.96%	103.81%	100.34%	97.99%
	Original	283.80	319.49	385.52	621.71
10^4	*hipec*	279.29	321.39	384.27	616.53
	Ratio	98.41%	100.59%	99.68%	99.17%
	Original	1656.75	2344.04	3393.79	6008.17
10^5	*hipec*	1621.13	2299.00	3328.06	5917.32
	Ratio	97.85%	98.08%	98.06%	98.49%
	Original	15279.53	22338.70	33497.12	60313.42
10^6	*hipec*	13953.89	19997.67	28907.42	51296.06
	Ratio	91.32%	89.52%	86.30%	85.05%

Table VII. Total elapsed time of SMPEG and DBG

	Allocated page frame size (MB)			
	64	48	32	16
	Elapsed Time (in seconds)			
Mach LRU	1133.72	1193.92	1236.53	1349.26
hipec FIFO	1087.33	1095.50	1137.64	1169.91
Ratio	95.91%	91.76%	92.00%	86.71%

Overall improvements in system performance

The previously presented benchmarks were used to assess the overall improvement in system performance. When running under the *hipec* system, all applications, except the DBG generator, ran as unprivileged applications. The concurrently run applications included the NLJ operator (**J**) with a 60 MB outer relation and a 4 KB inner relation, the RDR operator (**R**) for randomly accessing 10^5 tuples, the SMPEG video player (**S**) for playing 129.58 MB MPEG video data at an averaged 31.16 KB/sec, access rate, and finally, the sequential-write DBG generator (**G**) for building a 60 MB database. As shown in Table VIII, the elapsed time reductions ranged from 7–40% for different application combinations.

Ill-advised applications

The *hipec* system recovery scheme protects applications from any ill-advised application (i.e. applications that make *wrong* caching decisions.). Two issues related to the recovery scheme were investigated in this experiment. The first was whether the recovery scheme helped the ill-advised application in reducing the number of page replacements. The second was to find out whether the recovery scheme protects applications from inducing extra page replacements when these applications ran concurrently with an ill-advised application. The RDR operator with the MPFIFO2 caching strategy was used to explore these issues. The MPFIFO2 caching strategy was modified to act like an ill-advised caching strategy by replacing the page frames with the highest priorities. The ill-advised RDR operator ran as an unprivileged application and randomly retrieved 10^5 and 10^6 tuples in each evaluation. The system was booted with 12 and 16 MB of physical memory for the evaluations.

By comparing the rows *hipec+ill-advised-recovery* and *hipec+ill-advised+recovery* in Table IX, the recovery scheme is seen to have been helpful in reducing the elapsed time and the number of block I/Os for the ill-advised RDR. Without a recovery scheme, wrong caching decisions always keep low-priority data cached. Because low-priority data has little possibility of being re-accessed, keeping them cached will cause most of the page frames to be occupied by rarely used data and reduce the page-frame utilization. As a result, more page-replacement operations are induced. The recovery scheme recovered the ill-advised caching decisions such that page-frame utilization was not reduced. The RDR operator, however, still suffered from wrong caching decisions, because they induced recovery processing overhead and extra disk I/O operations. The performance difference in the RDR was not significant when the number of accessed tuples was small (i.e. 10^5).

Table X shows how the recovery scheme protected other applications from an ill-advised

Table VIII. Elapsed time for concurrently run multiple applications

Application combination		Allocated page frame size (MB) 64	48	32	16
		Elapsed time (in minute)			
J+S	Original	279.67	285.31	289.44	344.46
	hipec	174.07	206.25	251.67	317.84
	Ratio	62.24%	72.29%	86.95%	92.27%
R+S	Original	118.82	128.21	147.95	216.99
	hipec	108.91	114.07	126.56	176.14
	Ratio	91.66%	88.97%	85.54%	81.17%
J+R+S	Original	316.10	353.07	374.68	489.66
	hipec	211.17	263.24	313.14	408.35
	Ratio	66.80%	78.56%	83.58%	83.39%
J+G	Original	245.81	251.67	267.33	274.61
	hipec	148.61	166.14	198.44	252.59
	Ratio	60.46%	66.02%	74.23%	91.98%
R+G	Original	84.06	101.13	119.68	232.77
	hipec	67.79	83.81	96.68	181.15
	Ratio	80.64%	82.87%	80.78%	77.82%
J+R+G	Original	278.90	314.17	356.39	444.50
	hipec	173.37	229.31	284.16	397.93
	Ratio	62.16%	72.99%	79.73%	89.52%
J+R+S+G	Original	379.95	422.59	456.08	572.30
	hipec	245.51	303.10	341.60	482.79
	Ratio	64.63%	71.72%	74.90%	84.36%

application. In this experiment, the ill-advised RDR operator ran concurrently with other applications, including the NLJ operator, the SMPEG player and the DBG generator. When running under the *hipec* system, only the RDR operator ran as an unprivileged *hipec* application. Other applications ran as a general application that did not invoke any *hipec* services. The system was booted with 64 MB for the evaluation.

Table IX. Evaluations of ill-advised RDR operator

No. of accessed		Page frame size (MB) 12		16		12	16
		Elapsed time (in sec/normalized)				No. of block I/O	
10^5	original Mach	8916.42	1.0000	6008.17	1.0000	203833	137287
	hipec+ill-advised-recovery	9415.34	1.0600	6250.99	1.0404	211267	141509
	hipec+ill-advised+recovery	9118.82	1.0227	6154.77	1.0244	203959	137774
10^6	original Mach	98114.17	1.0000	60313.42	1.0000	2242611	1378580
	hipec+ill-advised-recovery	117432.85	1.1969	68522.28	1.1361	2580348	1549661
	hipec+ill-advised+recovery	108043.62	1.1012	65633.67	1.0882	2423375	1474116

Table X. Evaluating the system performance with an ill-advised application

	Applications					
	J	M	G	J	M	G
booted with 64 MB	Elapsed time (in min)			No. of block I/O		
under Mach	379.83	93.42	6.98	67515	33173	15360
under hipec-recovery	491.70	104.41	9.67	74238	33173	15360
under hipec+recovery	440.77	101.35	9.06	67916	33173	15360

Evaluations results show that the recovery scheme did limit the impact of the ill-advised RDR operator. Without the recovery scheme, the number of block I/Os from the NLJ operator was greatly increased. When the recovery scheme was enabled, the NLJ operator caused approximately the same number of block I/Os as when evaluated under the Mach kernel.

However, the elapsed time of the NLJ was increased by the ill-advised RDR operator. The elapsed time reduction of the NLJ operator was due to the heavier loading on the disk device such that the NLJ operator had to wait for the device. In addition, the ill-advised RDR took longer to finish its job and occupied page frames for longer periods. The occupation caused more page-frame competition, degrading the performance of the NLJ operator. Similar results from the SMPEG player and the DBG generator were also seen. These experimental results hint that the system must have a protection mechanism that revokes *hipec* services for ill-advised applications. An ill-advised application can be detected by the recovery scheme if it is monitored for excessive wrong caching decisions.[14] Revocation management is an interesting topic that requires further research to decide on threshold numbers for judging when an application is ill-advised.

RELATED WORK

Many research prototypes and development systems have addressed memory caching problems. Mach[8] exports the EMM interface to manage the paging operations of each memory-mapped region, but it lacks an interface for supporting application-specific cache management. The work of PREMO[6] employs message-passing IPC to extend the EMM interface to accommodate user-level page-replacement policies. The referenced and modified bits (R&M) for page frames can be obtained from the kernel by invoking new system calls, but only the R&M information can be retrieved in their implementation. Another work, called the POD,[7] further extends the Mach system to support user-level physical-memory management using message-passing IPC. In that implementation, applications can directly access kernel resources, but only trusted applications are allowed to invoke the POD interface. Neither PREMO nor POD address any system performance considerations and page-frame reclamation policy.

V++[20] uses the Segment Manager (SM) to manage physical memory, and has an interface to migrate page frames among different segments. A Memory Market (MM) approach[21] is used to allocate page frames among applications. All the operations of V++ involve transferring control among address spaces using IPC, such that huge IPC communication overhead is expected. The system performance of V++ is not addressed in its literature. The recent Cache Kernel questions the flexibility of existing microkernel operating systems.[22] Cache kernel proposes a new kernel architecture that allows applications to build their own specific

Application Kernel (AK) to meet their needs. Cache kernel uses memory-mapping IPC to communicate with the AK by loading and unloading kernel-object descriptors. To context-switch the AKs is time-consuming in loading and in unloading huge numbers of kernel-object descriptors.

The *SPIN*[23] system can dynamically extend system services to meet application requirements. Specific applications can dynamically load object codes into the kernel and link them to specialize the system to meet their needs, which would create the least overhead in system extension. *SPIN* guarantees system safety by using a secure programming language, Modula-3, to implement any extended service. The loaded object codes can access kernel resources directly by using unforgeable capabilities. Another development system, the Exokernel,[24,25] uses *software fault isolation* techniques[26] to export low-level hardware services to applications. Conventional operating system services, such as virtual-memory management and IPC, are implemented as libraries that can easily be specialized to meet application needs. *SPIN* and Exokernel are both targeted at providing applications with fine-grained management of system resources. Applications can directly manage hardware resources, such as the processor, the interrupts and the physical memory, etc.

The work of two-level file cache management,[13,14] built on top of the conventional file system, allows applications to select page replacement policies for each opened file, and introduces the *LRU-SP* policy to fairly share page frames among applications. Since conventional file systems do not support multiple caching buffers for each single file, simultaneously sharing a file among applications with different replacement policies is not supported in their implementation.

There are network studies[27,28,29] that use the interpretation approach in packet demultiplexing. Applications can program their packet filters using the filter language. When any packet comes into the system, the network driver interprets the filter and forwards the packets to its destination application. The interpretation overhead is negligible as compared to the long latency time of network I/O operations.

CONCLUDING REMARKS

It is impossible to implement one particular caching strategy within a kernel that can adequately serve all applications. The goal of the *hipec* system is to make use of application knowledge to reduce the number of page faults. Since page faults usually cause disk I/O operations, reducing the number of page faults means reducing disk I/O operations, Thus application performance and system throughput is increased.

The *hipec* system supports applications accessing data by memory-mapping the data into the address space of each application. Since data are cached in the address space of each application, even when a file is shared, each application can implement its own specific caching strategy to manage its memory-mapped region without any conflicts.

Hipec employs the FIFO2-MR policy to fairly share page frames among applications to reduce the impact from ill-advised applications. Auxiliary tools in the *hipec* implementation can help application designers identify performance bottlenecks and prepare intelligent caching strategies. Based on empirical results, the *hipec* system has proven to be applicable and able to achieve the claimed expectations.

However, a concern about the *hipec* system is whether there is significant benefit in employing the in-kernel strategy interpretation scheme. The answer to this concern is positive. The interpretation scheme is simpler than any domain-crossing approaches that can reduce the kernel complexity. This approach is also efficient in supporting applications to manage their

virtual memory cache without domain-crossing overhead. An experiment was used to explore the significance of strategy interpretation by evaluating the RDR benchmark. By concluding from the results listed in Table IV, the strategy interpretation was proven to create a significant performance gain to application performance. In addition, since the page-frame management operations are performed by the kernel, each operation performed is trustworthy. Protecting systems from any misbehaving application will be easier than with advanced techniques that use secure programming languages.

Whether the current *hipec* commands are sufficient to implement any specific caching strategy is another concern for *hipec* implementation. Since *hipec* commands are derived from basic virtual-memory management operations, they can currently be used to implement any caching strategy that conventional kernels can. However, the *hipec* command set does not include any commands related to data prefetching.[30] The purpose of data prefetching is to initiate I/O requests as soon as possible, so that I/O latency can be reduced by parallelizing application computation and data fetching. The performance gain resulting from data prefetching, as reported by other research,[31] is dominated by the disk-scheduling policies, and will be another interesting topic, but is beyond the scope of this paper.

REFERENCES

1. S. R. Das and R. M. Fujimoto, 'An adaptive memory management protocol for time warp parallel simulation', *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994, pp. 201–210.
2. D. Rotem and J. L. Zhao, 'Buffer management for video database systems', *Proc. Eleventh International Conference on Data Engineering*, Taipei, Taiwan, ROC, March 1995, pp. 439–448.
3. M. Stonebraker, 'Operating system support for database management', *Communications of the ACM*, **24**(7), 412–418 (1981).
4. S. J. White and D. J. DeWitt, 'QuickStore: a high performance mapped object store', *Proc. ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994, pp. 395–406.
5. K. Krueger, D. Loftesness, A. Vahdat and T. Anderson, 'Tools for the development of application-specific virtual memory management', *Proc. ACM Eighth Annual Conference On Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, September 1993, pp. 48–64.
6. D. McNamee and K. Armstrong, 'Extending the Mach external pager interface to accommodate user-level page replacement policies', *Proc. First USENIX Mach Workshop*, Burlington, VT, USA, October 1990, pp. 17–29.
7. S. Sechrest and Y. Park, 'User-level physical memory management for Mach', *Proc. USENIX Mach Symposium*, Monterey CA, November 1991, pp. 189–200.
8. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, 'Mach: a new kernel foundation for UNIX development', *Proc. 1986 Summer USENIX Conference*, Atlanta, GA, USA, July 1986, pp. 93–112.
9. M. Young, A. Tevanian, R. Rashid, D. Gloub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, 'The duality of memory and communication in the implementation of a multiprocessor operating system', *Proc. Eleventh ACM Symposium on Operating System Principles*, Austin, TX, November 1987, pp. 63–76.
10. T. E. Anderson, B. N. Bershad, E. D. Lazowska and H. M. Levy, 'Scheduler activations: effective kernel support for the user-level management of parallelism', *ACM Transactions on Computer and Systems*, **10**(1), 53–79 (1992).
11. J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.
12. K. Bala, M. F. Kaashoek and W. E. Weihl, 'Software prefetching and caching for translation lookaside buffers', *Proc. First Symposium on Operating Systems Design and Implementation*, Monterey, CA, USA, November 1994, pp. 243–253.
13. P. Cao, E. W. Felten and K. Li, 'Application-controlled file caching policies', *Proc. USENIX SUMMER 1994*

Technical Conference, Boston, MA, USA, June 1994, pp. 171–182.

14. P. Cao, E. W. Felten and K. Li, 'Implementation and performance of application-controlled file caching', *Proc. First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994, pp. 165–178.
15. R. P. Draves, 'Page replacement and reference bit emulation in Mach', *Proc. USENIX Mach Symposium*, Monterey CA, USA, November 1991, pp. 201–212.
16. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
17. H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.
18. H. T. Chou and D. J. DeWitt, 'An evaluation of buffer management strategies for relational database systems', *Proc. 11th International Conference on Very Large Data Bases,* Stockholm, Sweden, August 1985, pp. 127–141.
19. E. J. O'Neil and P. E. O'Neil and G. Weikum, 'The LRU-K page replacement algorithm for database disk buffering', *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993, pp. 297–306.
20. K. Harty and D. R. Cheriton, 'Application-controlled physical memory using external page-cache management', *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp. 187–197.
21. D. R. Cheriton and K. Harty, 'A market approach to operating system memory allocation', *Technical Report*, Stanford University, 1992.
22. D. R. Cheriton and K. J. Duda, 'A caching model of operating system kernel functionality', *Proc. First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994, pp. 179–194.
23. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers and S. Eggers, 'Extensibility, safety and performance in the *SPIN* operating system', *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995, pp. 267–284.
24. D. R. Engler, M. F. Kaashoek and J. W. O'Toole Jr., 'The operating system kernel as a secure programmable machine', *ACM Operating Systems Review*, **29**(1), 78–82 (1995).
25. D. R. Engler, M. F. Kaashoek and J. O'Toole Jr., 'Exokernel: an operating system architecture for application-level resource management', *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995, pp. 251–266.
26. R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham, 'Efficient software-based fault isolation', *Proc. Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, November 1993, pp. 203–216.
27. S. McCanne and V. Jacobson, 'The BSD packet filter: a new architecture for user-level packet capture', *Proc. Winter 1993 USENIX Conference*, San Diego, CA, January 1993, pp. 259–269.
28. J. C. Mogul, R. F. Rashid and M. J. Accetta, 'The packet filter: an efficient mechanism for user-level network code', *Proc. Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, November 1987, pp. 39–51.
29. M. Yuhara and B. N. Bershad, 'Efficient packet demultiplexing for multiple endpoints and large messages', *Proc. 1994 Winter USENIX Conference*, San Francisco, CA, January 1994, pp. 153–163.
30. P. Cao, E. W. Felten, A. Karlin, and K. Li, 'A study of integrated prefetching and caching strategies', *Proc. International Conference on Measurement and Modeling of Computer Systems*, Ottawa, Canada, May 1995, pp. 188–197.
31. P. Cao, E. W. Felten, A. R. Karlin and K. Li, 'Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling', *Technical Report No. TR-CS-95-493*, Department of Computer Science, Princeton University, 1995.
32. A. W. Appel and K. Li, 'Virtual memory primitives for user programs', *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp. 96–107.
33. P. C. H. Lee, M. C. Chen and R. C. Chang, 'Hipec: high performance external virtual memory caching', *Proc. First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994, pp. 153–164.