In-Kernel Policy Interpretation for Application-Specific Memory Caching Management[*]

Paul C. H. Lee[†] Meng Chang Chen[‡] Ruei-Chuan Chang[§]

Department of Computer and Information Science[†§]
National Chiao Tung University
Hsinchu, Taiwan 30050, R.O.C.

Institute of Information Science[‡§]
Academia Sinica, Taipei 11529, R.O.C.

[*]This research work was partially supported by the National Science Council of Republic of China under grants No. NSC83-0408-E-001-008 and NSC84-2213-E-001-009.

[†]Paul C. H. Lee is a full time Ph.D student at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

[‡]Meng Chang Chen is with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.

[§]Ruei-Chuan Chang is with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C, and the Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.

SUMMARY

Traditional operating systems manage the page frame pool with the LRU-like policies which cannot properly serve all types of memory access patterns of various applications. As a result, many memory-intensive applications induce excessive page faults and page replacements when running with access behaviors other than the LRU-like patterns. This paper presents a **hi**gh **p**erformance **e**xternal virtual memory **c**aching mechanism (*hipec*) to allow user applications to run with their own specific page frame management policies. The user applications inform the operating system of their specific management policies by loading a set of macro-like commands to the kernel. When page faults or page replacements happen, the operating system will interpret the commands and perform the corresponding specific management policies. The empirical evaluations show that the *hipec* mechanism induces little overhead and significantly increase the application and system performance.

Keywords: *hipec*; resource management; page replacement; caching; *FIFO2-SP*;

1 INTRODUCTION

The advances in computer technologies have increased the processor speed several times in the passed decade. However, the disk I/O performance is little improved to catch up with the processor speed. As a result, the main memory is used as the buffer cache to reduce the performance gap between the processor and the storage device. Traditionally, the operating system manages the buffer cache pool with the LRU-like policies [10] which can serve applications in time-sharing environments with satisfactory performance, but not for those specific, memory-intensive applications, such as the database management systems [24], the multimedia applications [20] and the scientific simulators [17]. These specific applications with access behaviors other than the LRU-like patterns would induce excessive page faults and page replacements when running on top of existing operating systems. Since page replacement operations usually involve disk I/O activities, the performance of these memory-intensive applications degrades.

The mismatch between the application access behaviors and the operating system page frame management policy can be improved by delegating the page frame management policies to user applications. Since the user applications know their access behaviors, when page faults or page replacements happen, the user applications can decide to return the least important page frames. Traditionally, such delegating mechanism is implemented in Upcall [2, 13] or IPC [18, 22] by transferring control from the operating system kernel to the user applications. These traditional *domain-crossing* techniques creates evident overhead in transferring control and passing messages, which compromise system performance.

In this paper, we challenge the necessity of domain crossing for delegating the page frame management policies to applications. A **hi**gh **p**erformance **e**xternal virtual memory **c**aching mechanism (*hipec*) [14, 15] is proposed to allow user applications to prepare and perform their own specific page frame management policies. By loading a set of macro-like commands to the operating system, the user applications inform the operating system of their specific management policies. When page faults or page replacements happen, the operating system will interpret the commands and perform the corresponding specific management policies to suit the access patterns of user applications. *Hipec* does not create any domain-crossing overhead which is more efficient than the traditional domain-crossing techniques.

2 MOTIVATION AND DESIGN ISSUES

When compared to the traditional in-kernel page frame management mechanism, the overhead created from the domain-crossing techniques can be characterized as follows. First, transferring control from kernel to user applications needs to save the current kernel mode context and load the new user mode context. The overhead includes saving and loading the register set, changing the stacks, passing arguments, and executing the specific management routines. Second, the applications need to invoke system calls to obtain system status in making specific page frame management decisions, since direct accesses from user applications of the kernel data structures are prohibited. For example, when determining the dirty page frames to be flushed, the user applications need to invoke system calls to get the modified bits information [18]. Third, extra scheduling overhead is generated if the operating system has to reclaim page frames from a bundle of specific applications. Previous implementations [18, 22] extending page frame management policies created 10% to 14% overhead that follow the above observations.

Unlike the domain-crossing techniques, the *hipec* mechanism uses the in-kernel policy interpretation approach to extend the page frame management policies. Figure 1 illustrates the sketch of the *hipec* mechanism. Initially, (1) each specific policy, represented by a set of macro-like commands, is loaded into the kernel. Later on, when any page fault happens, as (2), the page fault handler asks the frame manager for a free page frame, shown in (3). This request is satisfied immediately when there remain free page frames in the system. If the free page frames are exhausted, the frame manager starts to reclaim page frames from the user applications. The frame manager selects victim applications and (4) interprets their specific management policies to decide the page frames to be reclaimed. When enough page frames are reclaimed, the frame manager stops the reclamation operations and (5) transfers control to the originally faulted application. This design has several advantages:

- **Performance.** Application performance and overall system throughput are increased, if all the applications know their access patterns and manage their resources accordingly for which *hipec* is intended. As there is no domain-crossing in the *hipec* mechanism, the overhead is nominal.

- **System safety.** The specific page frame management policies of user applications, written in the *hipec* commands, are performed by the operating systems. The kernel resources are protected from direct accesses of user

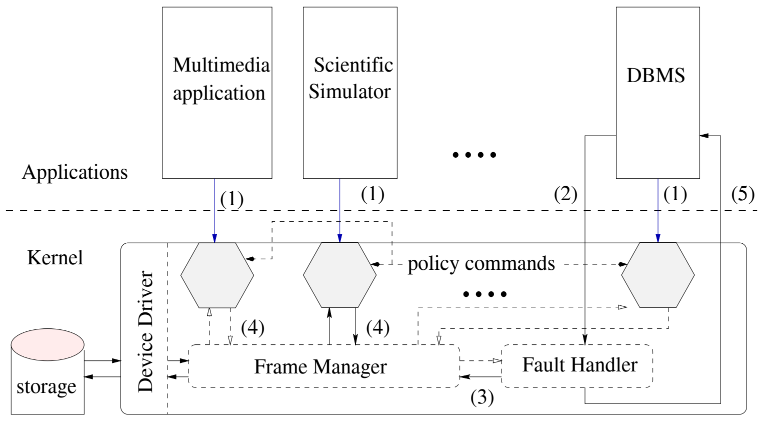


Figure 1: The proposed *hipec* approach.

applications which keeps the system safe from malicious applications. Additional mechanisms of the *hipec* implementation further protect the operating system from misbehaved policies. The detail of these protection mechanisms are described in Section 4.

- **Generality.** The set of *hipec* commands can be treated as an interface between the user applications and operating systems. The hardware architecture and the detail of the operating system internals are shielded from the application designers when program their specific page frame management policies. The applications and their specific page frame management policies can be ported to other systems support *hipec*.

- **Flexibility.** Though the number of current *hipec* commands is small, various policies can be implemented using the commands without any problem. The *hipec* approach is flexible as additional commands can be added into the *hipec* command set. In addition, a language-like pseudo code translator is implemented to provide convenience and further flexibility for the application designers.

We present the design and implementation of *hipec* in Section 3. The mechanisms of *hipec* to keep the system safe are described in Section 4. Section 5 shows the performance evaluations. We review the related works in Section 6 and conclude the paper in Section 7.

3 DESIGN AND IMPLEMENTATION

The *hipec* implementation is built on the OSF/1 MK5 operating system[1]. *Hipec* works with the Mach external memory management (EMM) interface to fully support application-controlled external memory caching management. With this extension, user applications can control the paging activities of virtual memory-mapped data via the EMM interface and handle the caching management of that virtual memory region. However, the concept and implementation of *hipec* are independent of EMM interface and can be easily applied to other operating systems.

3.1 Overview

The primary components of *hipec* implementation are the *hipec* commands, the application-specific page frame management policies and the *hipec* container. They are described respectively in the following sections.

Two kinds of *hipec* interface are exported to the user applications. One is for the privileged applications and the other is for the general (i.e. unprivileged) applications. For the privileged invocations, the frame manager allocates the requested size of page frames to the applications. Before the privileged application is terminated, no page frame reclamation is allowed to the allocated page frames. If the remaining free page frames cannot satisfy the privileged requests, an error indicator is returned to the applications. For the unprivileged applications, the frame manager will not allocate any page frame to them until page faults happen, and the page frames are subject to be selected for reclamation. When the system

[1]OSF/1 MK5 is a microkernel operating system that uses the Mach *nmk15.0* kernel and runs the OSF/1 5.0.2 server.

No.	Command	Binary	Operations
1.	Return	00000000	The end of execution.
2.	Arith	00000001	Arithmetic operations for Integer.
3.	Comp	00000010	Comparison operations for Integer.
4.	Logic	00000011	Logical operations.
5.	EmptQ	00000100	Test if the specified queue is empty?
6.	InQ	00000101	Test if the specified page is in the specified queue?
7.	Jump	00000110	Branch to next command.
8.	Dequeue	00000111	Move the specified page from a specified queue.
9.	EnQueue	00001000	Add a specified page to one queue.
10.	Request	00001001	Request page frames from the system.
11.	Release	00001010	Release page frames to the system.
12.	Flush	00001011	Flush page content to disk.
13.	Set	00001100	Set or reset the referenced or modified bits.
14.	Ref	00001101	Test if specified page is referenced?
15.	Mod	00001110	Test if specified page is modified?
16.	Find	00010000	Find the specified page if giving the virtual address?
17.	Activate	00010010	Invoke another policy event.

Table 1: The *hipec* command set.

free page frames are exhausted, the frame manager will select the victim applications to reclaim the page frames and perform page replacements. The page frame reclamation policy is described in Section 3.5.

3.2 *Hipec* commands

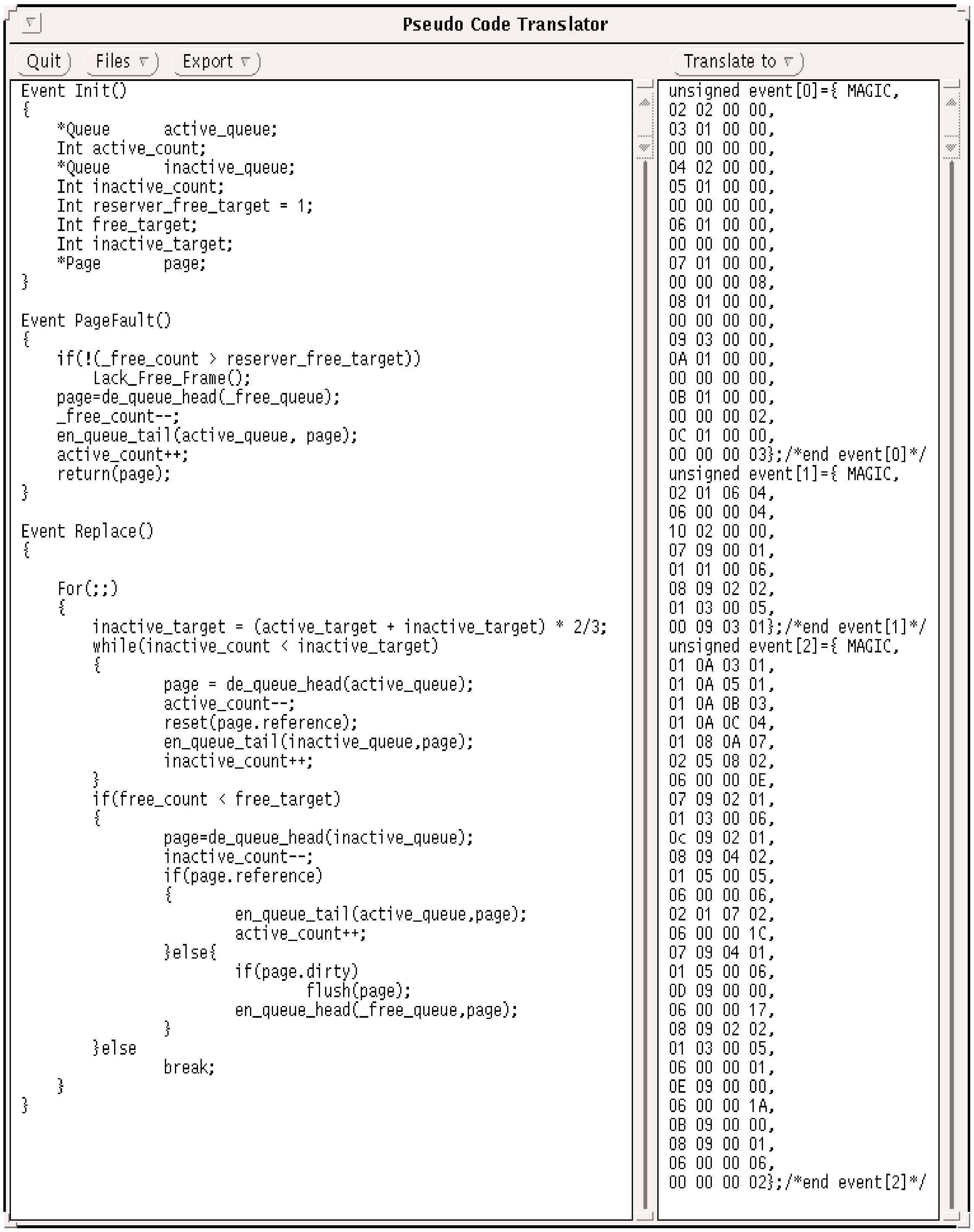
The *hipec* commands are a set of 32-bits commands which consist of an 8-bits operator code and up to two operands. There are four data types for the operand variables, the *Integer*, **Integer*, **Page* and **Queue*[2]. Each command is implemented as a macro or a procedure call of the virtual memory management operations, such as adding a page frame to a specified queue or flushing page frames. The details of operating system internals are shielded behind the exported *hipec* commands. The application designers program their page frame management policies in the *hipec* commands and need not pay attention to the kernel implementations. Table 1 lists the currently implemented *hipec* commands. The detail syntax and usage of the *hipec* commands are described in the *hipec* document [15].

[2] The '*' means the pointer to the specified data type.

3.3 Application-specific page frame management policy

The application-specific page frame management policies are programmed in the *hipec* commands. Each policy contains at least three mandatory segments of policy commands, named the **Initial** event, the **PageFault** event and the **Replace** event. When the *hipec* interface is invoked, the commands of the Initial event are interpreted to initiate the operand variables by building a new kernel data structure, called the *container*, to store the operand variables. The container is introduced in next section.

The PageFault segment is interpreted when page fault occurs. After the page fault handler grants a page frame and maps the physical address to the faulted address, the PageFault event is interpreted by the page fault handler to perform housekeeping operations, such as recording the physical and virtual address of the mapped page frame for future uses. The Replace event is interpreted when the application is requested to return page frames. Applications use the Replace event to select the least important page frames, according to their access patterns. to be reclaimed. As the allocated page frames of the privileged applications can not be reclaimed by the operating systems, the Replace event of privileged applications is invoked by the PageFault to select one of its allocated page frames to locate to the faulted address, when the allocated free page frames are exhausted. Alternately, for the unprivileged applications selected to return page frames, the Replace event is interpreted by the frame manager to reclaim page frames and do page replacement operations. In addition to the mandatory events, application may define 256 events in maximum. The non-mandatory events are used as subroutines and are invoked by other events by the *Activate* command. A user level pseudo code translator is implemented to help the application designers with designing the high level, language-like page frame management policies. Figure 2 is an example of the specific page frame management policy in the pseudo code and translated code.



```
Event Init()
{
    *Queue      active_queue;
    Int active_count;
    *Queue      inactive_queue;
    Int inactive_count;
    Int reserver_free_target = 1;
    Int free_target;
    Int inactive_target;
    *Page       page;
}

Event PageFault()
{
    if(!(_free_count > reserver_free_target))
        Lack_Free_Frame();
    page=de_queue_head(_free_queue);
    _free_count--;
    en_queue_tail(active_queue, page);
    active_count++;
    return(page);
}

Event Replace()
{

    For(;;)
    {
        inactive_target = (active_target + inactive_target) * 2/3;
        while(inactive_count < inactive_target)
        {
                page = de_queue_head(active_queue);
                active_count--;
                reset(page.reference);
                en_queue_tail(inactive_queue,page);
                inactive_count++;
        }
        if(free_count < free_target)
        {
                page=de_queue_head(inactive_queue);
                inactive_count--;
                if(page.reference)
                {
                        en_queue_tail(active_queue,page);
                        active_count++;
                }else{
                        if(page.dirty)
                                flush(page);
                        en_queue_head(_free_queue,page);
                }
        }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 event[0]*/
unsigned event[1]={ MAGIC,
02 01 06 04,
06 00 00 04,
10 02 00 00,
07 09 00 01,
01 01 00 06,
08 09 02 02,
01 03 00 05,
00 09 03 01};/*end event[1]*/
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,
01 03 00 05,
06 00 00 01,
0E 09 00 00,
06 00 00 1A,
0B 09 00 00,
08 09 00 01,
06 00 00 06,
00 00 00 02};/*end event[2]*/
```

Figure 2: Example of implementing the First In First Out with Second chance page frame management policy.

3.4 *Hipec* container

On the Mach operating system, the virtual address space of each application is divided into a set of virtual memory regions that each can be represented as a VM object. A VM object can be a memory-mapped data file or a segment of address space with the same protection attributes. A new kernel object, called the *hipec container*, is introduced that is mounted under the corresponding VM object that requires specific page frame management policy to store the *hipec* related information. The container data structure is allocated from the zone system [21].

The *hipec* container is primarily used to store the operand variables of *hipec* commands. The operand variables and the pointers to the operand variables are stored in the *operand array* of the container. When the *hipec* invocation is initialized, the *hipec* commands are loaded into the operating system kernel and the Initial event is interpreted to allocate the operand variables from the *hipec* container and assign the index number to the operand array for each operand variable. Figure 3 shows the relationship among the *hipec* commands, the container and the operand variables.

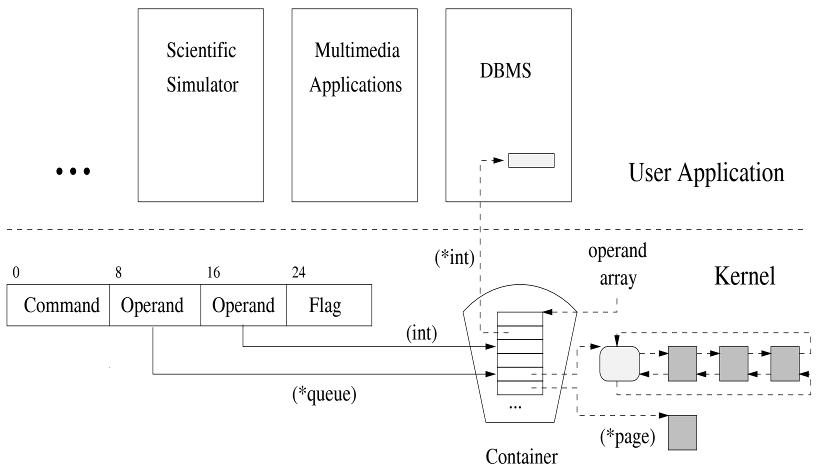


Figure 3: The *hipec* container.

```
vm_pageout_scan(){
        while(vm_page_free_count < vm_page_free_target) {
                page = dequeue(vm_page_inactive_queue);
                if(page->referenced ||
                  pmap_is_referenced(page->phys_addr)) {
                        enqueue(vm_page_active_queue, page);
                } else {
                    if (page->hipec == TRUE) {
                            set_timeout(timeout_detecting_function, TimeQuantiumOfScheduling);
                            reclaim_page = interpreter(current_task->map->object->container->Replace);
                            reset_timeout(timeout_detecting_function);
                            if(reclaim_page != page) {
                                    swap_location(*page, *reclaim_page);
                                    insert_holder(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: The *FIFO2-SP* page frame reclamation policy. The *FIFO2-SP* policy is modified from the *FIFO2* policy. The added components are boxed by the dashed line.

3.5 Page frame reclamation policy

Mach kernel uses the *FIFO*2[3] policy to manage the system page frame pool by maintaining the *active*, *inactive* and *free* queues [10]. Page frames that are allocated to user applications are linked either to the *active* or the *inactive* queues, while the free page frames are linked to the *free* queue. Each time as the free page frames are exhausted, the frame manager is waken up to reclaim pages from the head of the *inactive* queue. Page frames that have been referenced since the last page reclamation activity get the second chance and still reside in the *inactive* queue. Otherwise, the victim pages are reclaimed and moved to the *free* queue.

The *FIFO*2-*SP* page frame reclamation policy

Hipec uses the *FIFO*2-*SP*[4] policy, modified from the *FIFO*2 policy in the Mach operating system, for the frame manager to select victim unprivileged applications[5] to reclaim page frames. The *FIFO*2-*SP* policy is listed in Figure 4 and is described as follows. When to reclaim page frames, the frame manager is waken up to examine the page frames in the same order as the original *FIFO*2 policy, i.e. page frames in the head of the *inactive* queue are selected as the victim page frames. However, the *hipec* frame manager does not directly reclaim the victim page frame; rather it select the application owns the page frame as the victim application. The frame manager will interpret the Replace commands of the victim application and reclaim the page frame suggested by the Replace commands.

Swapping locations of the selected page and the reclaimed page

Before reclaiming the application-suggested page, the frame manager-selected victim page is removed from the *inactive* queue and inserted to the location of the reclaimed page. This movement is called *Swapping* that prevents the same page

[3] First In First Out with Second chance page frame reclamation policy.

[4] The First In First Out with Second chance, Swapping page locations and Holder policy.

[5] Page frames belonging to privileged *hipec* applications will not appear on the *FIFO*2 queues. Those page frames are under the control of the privileged applications.

frame repeatedly selected as victim page and the same application repeatedly selected for reclaiming page frames. Swapping locations makes the *FIFO2-SP* policy as fair as the original Mach *FIFO2* policy.

Building holder to record reclamation decision

The reason to have specific applications to decide the page frames to be reclaimed is that applications should know their access patterns better. However, application designers might make mistake in designing the page frame reclamation policy. I.e. the application-suggested reclaimed page will be referenced before the frame manager-selected victim page, which should have been replaced. Two major problems occur due to the bad replacement decisions.

- The bad application-specific page replacement decision will incur extra page faults to that application that cause the frame manager to allocate extra free page frames to that application. This problem should be avoided since the system resource should not be monopolized by some applications.
- When the number of remaining free page frames is small, the excessive page faults and page frame allocations tends to incur the page reclamation operations. As a result, other applications with pages in the head of the *inactive* queue will be requested to return pages and are influenced or penalized for the bad page frame replacement decisions of other applications.

Above mentioned problems can be avoided by introducing the *holder* table. A *holder* is a data structure used to record the discrepancy between the frame manager-selected page and the application-suggested reclaimed page. When application-specific page replacement decision is different from the frame manager, a holder is built within the holder hash table. The virtual address of the reclaimed page is used as the hash key value that is recorded in the holder together with the pointer to the frame manager-selected page. Later on, when page fault happens, if the page fault handler finds a holder hashed by the faulted

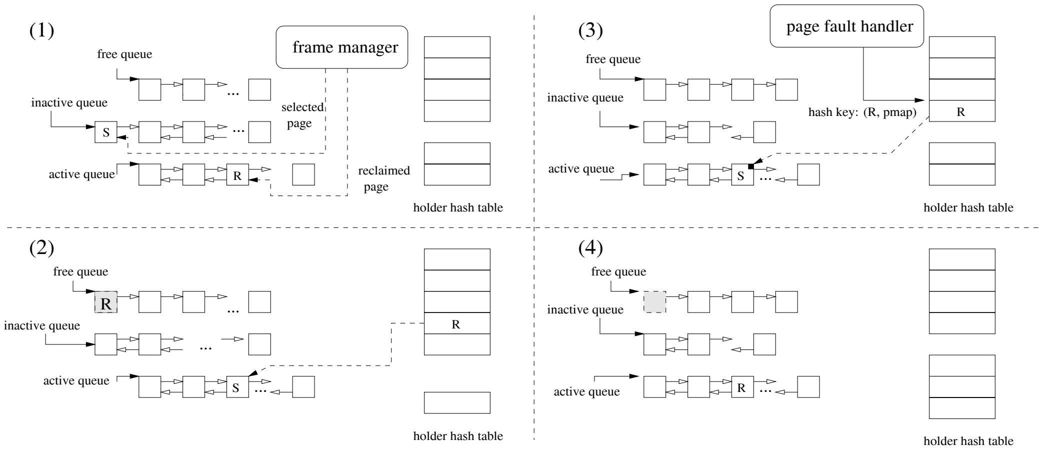


Figure 5: Example of using swapping and holder of the *FIFO2-SP* reclamation policy. (1) When the selected page(**S**) is to be reclaimed, the frame manager interprets the Replace event of the victim application and reclaims the **R** page instead. (2) **R** and **S** pages are swapped. Then, the **R** page is reclaimed into the free queue. In addition, a *holder* is inserted into the holder hash table to indicate the exchange. (3) Later on, when page fault happens, the page fault handler will check the holder of the faulted address. (4) If the faulted virtual address **R** is found in the hash table and the referenced bit of the pointed page(**S**) has not been set, the page fault handler amends the previous bad decision by reclaiming the **S** page frame and mapping that physical page frame to the faulted virtual address (**R**).

address, it checks the referenced bit of the page pointed by the holder. If the pointed page is not accessed since the holder is built, the previous application-specific page replacement decision is wrong. The page fault handler will unmap the previous frame manager-selected page, map the physical page frame to the faulted address and clears the holder. Figure 5 gives an example of the swapping and holder mechanisms of the *FIFO2-SP* policy.

4 SYSTEM SAFETY

To keep the system safe is one of the fundamental functionalities of the operating system. *Hipec* is efficient in delegating the page frame management policies to user applications, nevertheless, *hipec* invocation of bad application-specific policies may crash or monopolize the system and, thus, induce safety problem. The safety problems and the solutions of the *hipec* approach are described as follows.

Syntax checking for dangling references

The operand fields of the *hipec* commands may have reference of non-existent variables or with wrong data type. Interpreting the erroneous commands will reference the wrong operands or destroy other kernel data structures. Our solution to avoid the dangling references is to check the syntax of the commands before loading the policy into the kernel. Since each operand variable is initialized before loading the *hipec* commands, the existence and data type of each operand can be easily checked.

Runtime checking

Though static syntax checking can avoid the dangling references, some dynamic errors are hard or even impossible to be discovered from the syntax checking. For example, one application may have the *Arith* command to divide an integer by an operand with zero value. Or an application may try to get a page frame from an empty queue by *DeQueue* command. In *hipec*, before executing each *hipec* command, its corresponding conditions are dynamically checked.

Timeout of policy execution

The other safety problem of *hipec* is long or infinite execution time for interpreting the application-specific policies. Since the policy is interpreted in kernel mode by the page fault handler or the frame manager, the interpretation operations cannot be preempted until finished. Consequently, a long or infinite policy interpretation, resulting from a badly written policy, will monopolize the system.

Two mechanisms are designed to prevent the problems. When a page fault happens, the page fault handler will interpret the PageFault event of the faulted application. Before starting to interpret any command, a timeout function is inserted into the clock timeout list. When the inserted timeout duration is expired, the clock interrupts the interpretation and sets the timeout flag in the container. The page fault handler checks the timeout flag before every command interpretation that it will detect the timeout, move to invoke the blocking routines to give up the processor and perform context switching. As the page fault handler runs in the context of the faulted application, the blocking invocation causes the faulted application to be blocked. The blocked application will be awaken to continue its policy interpretation in next scheduling time quantum and be arranged to the right location of the run queue by calculating the priority according to its system resource consuming. The timeout detection mechanism is active in each scheduling time quantum as long as the interpretation continues. The design limits the burden of long policy interpretation to the application itself only.

For the frame manager, the timeout detection mechanism is similar to that of page fault handler. When a page fault happens and the number of remaining free page frames is low, the page fault handler will wake up the frame manager to reclaim page frames. The frame manager starts to reclaim page frames following the *FIFO*2-*SP* policy to select a victim application. Then, interprets the Replace event of the application to reclaim page frames. An timeout function is inserted to the clock timeout list before policy interpretation. If the interpretation is not finished within the timeout duration, the frame manager will find out that the timeout flag is set and proceed to terminate the interpretation, reclaim the original selected page frame and record the reclamation information in the container of the long policy. The time spent in interpreting the commands are accounted for each application and is used in calculating the scheduling priority. After sufficient page frames are reclaimed, the frame manager wakes up the scheduler to select the next application to run. Currently, the timeout duration is set to a unit of scheduling time slice.

Flushing pages

As I/O operations are far slower than the processor, the CPU will be idle if a sequence of page flushing operations are interpreted since the interpreting routines are not preemptive.. *Hipec* solves this problem by reserving a small free page frame pool. When an application has pages to be flushed, the system trades the application with new free page frames from the reserved pool for the dirty pages.

Thus, the application needs not to wait for completion of the page flushing. The traded dirty pages are listed in a *FIFO*[6] queue which are flushed to the disk storage by the original Mach kernel pageout daemon.

5 PERFORMANCE EVALUATIONS

The performance of *hipec* is evaluated in three aspects. First, we measure the overhead created from *hipec* mechanism and give a comparison with previous domain-crossing techniques. Second, we compare the elapsed time for single applications running under the Mach kernel and the *hipec* mechanism. Third, multiple applications are running simultaneously to compare the system performance with and without the *hipec* mechanism. The *Acer Altos* 10000 machine is used as our experimental platform, which is an *Intel* 486-33 processor, *EISA* bus based machine with 128 Kbytes on-board cache memory. The primary storage devices are the *Segate* 31230N disks attached to the *AHA*-1742 *SCSI* adapter. The network adapter is *D-link ne*220*CT Ethernet* card.

5.1 Measure *hipec* Mechanism

First, we measure the page fault handling time for accessing 40 Mega bytes virtual address space under both the Mach kernel and the *hipec* environment. A experimental program is implemented to sequentially read/write 40 Mega bytes virtual memory of its addressing space. The read operation will cause the free page frames to be mapped to the specified virtual address space and does not cause any disk I/O activities, while the write operations will create dirty pages which will be paged out to the swapping storage during page replacement operation. The experimental program allocates 40 Mega bytes physical memory in the privileged *hipec* environment. When running under the Mach environment, there are 48 Mega bytes physical memory allocated under the Mach kernel[7]. To

[6]The First In First Out policy.

[7]When the OSF/1 MK is booted, the OSF/1 MK5 allocates about 5.94 Mega bytes physical memory in our platform to store the operating system and kernel data structures. In comparing

Evaluations	Averaged Time Overhead
40 Mega bytes sequential page fault without disk I/O activities (Read operation)	
Running under original Mach kernel	4016.5 msec
Running with *hipec* mechanism	4102.1 msec
hipec Overhead	2.13%
40 Mega bytes sequential page faults with disk I/O activities (Write operation)	
Running under original Mach kernel	82485.5 msec
Running with *hipec* mechanism	82675.2 msec
hipec Overhead	0.23%

Table 2: Measurement of *hipec* overhead.

make the comparison fair, the experimental program uses the same *FIFO2* page frame management policy when it invokes the privileged *hipec* service. Table 2 lists the evaluated elapsed time for the experimental program running under both the Mach and privileged *hipec* environments[8]. Since the experimental program running under both environments with the same *FIFO2* management policy, the difference of the evaluated time is considered as the *hipec* overhead including the time to interpret the *hipec* commands and the time for *hipec* operations, such as dynamic checking. The overhead of *hipec*, with or without disk I/O activities, is so small that can be compensated by reducing as few as one or two disk page I/O operations.

The *hipec* overhead is averaged[9] to compare with IPC or Upcall, listed in Table 3. Upcalls are implemented as procedure invocations from the kernel to user applications. The overhead is mainly in allocating new user stack and changing control to the stack. In Mach, the IPC mechanism is implemented by message passing. Messages are copied from the address space of the sender to the receiver. As we do not have Upcall implementation, the cost for the system call is used to represent the Upcall overhead. As described in previous sections, user applications using the domain-crossing techniques to implement their specific policies

with 40 Mega bytes in privileged *hipec* environment, allocating 48 Mega bytes physical memory in this experiment favors the evaluation under the Mach kernel which at least has 42 Mega bytes physical memory available for the experimental program.

[8]The time to initiate the invocation is not counted in the listed values.

[9]By "averaged", the overhead of *hipec* in Table 2 is divided by the total number of pages that is the averaged overhead per page incurred from *hipec* mechanism.

Evaluations	Averaged Time
averaged overhead for *hipec* approach	13.44 μ sec/page
Context Switch Overhead	31.7 μ sec
Single Null System Call	18.4 μ sec
Single System Call with 32 byte arguments	23.3 μ sec
Single Null IPC Call	292.3 μ sec
Single IPC Call with 32 byte arguments	317.9 μ sec

Table 3: Comparison of *hipec* to domain-crossing techniques.

need extra system call invocations to get information in deciding their page frame management decisions. The overhead of extending page frame management policies using Upcall or IPC is far larger than the values listed in Table 3. Because of the domain-crossing techniques usually requires context switching, the cost for context switching is also listed[10].

5.2 Performance evaluations of single applications

An instructional database management system (DBMS) and a MPEG video player are ported to the experimental platform to evaluate the *hipec* mechanism. Instead of using the traditional file I/Os, both applications are modified to use the virtual memory mapped I/O. Three most common access patterns [9] are presented in the evaluations.

Nested loop join operator

The join operator is one of the most important operations of relational database management systems. Traditional LRU-like policies select the least recently used pages to be replaced that do not match the sequential loop access pattern of the nested join operator and cause repeated page replacements. In this evaluation, the nested join operator runs as a *hipec* privilege application with a MRU-like application-specific policy. The inner relation of the nested join operator is 4 K bytes and the size of the outer relation is 60 Mega bytes. Each tuple in the relations is 64 bytes long. The evaluation is conditioned under different size of allocated page frames.

[10]The listed cost of context switching does not include the time to search through the run queue to find the next thread to run.

Data retrieval from binary index tree

Many data retrieval systems use the hierarchical index to speed up the data access of random stored data. The tree index file is accessed more frequently than the data files because each data retrieval accesses the tree index file first. Particularly, the ancestor nodes of the index tree have higher probability to be accessed than the descendant nodes. In this evaluation, the DBMS query randomly accessing data tuples from the binary tree index file is evaluated. The index file has 16 Mega bytes and the data file has 64 Mega bytes. Each index record is 16 bytes long consisting of primary index key, pointer to the data tuple and pointers to the left child and right child index records. Each data tuple is 64 bytes. The index tree is highly balanced with the height of 25.

The data retrieval query operator uses a simple management policy for the memory-mapped index file and the data file. Page frames are classified into different priority queues according to the index tree levels of the page frames. Queues of each level are managed with the *FIFO*2 policy. The queues with smaller index tree levels have higher priorities that will not be selected to return page frames when there exist page frames in the lower priority queues. The pages with level number larger than 22 and the data file pages are treated as the same priority. The elapsed times of the query randomly accessing sets of tuples of various size are recorded.

The sequential MPEG video player

Though a fully sequential access pattern can be served efficiently by the traditional LRU-like policy, we show here that the *hipec* implementation can be used to reduce the unnecessary competition of page frames of various applications. In this experiment, a software MPEG video player is used to play the MPEG video data. The video data file has 12.96 Mega bytes. A database data generator is running in background to build a 60 Mega bytes database. The MPEG player accesses the video data in sequential read pattern and the database generator is a sequential write application. The video player runs as a privileged application in *hipec* environment, uses a FIFO page frame management policy and is only allocated with 40 Kbytes physical memory. The database generator is unmodified for both environments.

Analysis

As the Mach kernel consumes about 6 Mega bytes physical memory, the size of available physical memory for user applications is 58, 40, 26 and 10 Mega bytes respectively when the Mach kernel is booted with 64, 48, 32 and 16 Mega bytes physical memory. To make the comparison fair, when running under the *hipec* environment, the applications are allocated 6 Mega bytes physical memory less when under *hipec* environment. Table 4, 5, and 6 list the elapsed time for applications running under the Mach and the privileged *hipec* environments. All the values are the average from 5 independent evaluations.

Nested join operator creates an interesting phenomenon. When the size of available physical memory is smaller than the data working set of the LRU-like management policy, regardless the size of available physical memory, the nested join operator causes repeated page replacements. Though the size of physical memory is varied from 64 Mega bytes to 16 Mega bytes, the elapsed time evaluated under the Mach kernel is close in each evaluation. The results in Table 4 show the phenomenon. When the nested join operator invokes the *hipec* interface to run the application-specific MRU-like policy to suit its access patterns, the performance is improved largely from 16% to 95%.

For the tree index query application, when the number of the random access records is small, the *hipec* approach gains little in reducing the number of page faults and page replacements. since most of the data can be cached in physical memory. Table 5 shows that the overhead of *hipec* mechanism does not slow down the application when record number is small, and the performance is increased by 9% to 15% when the access record is 1000000.

The evaluation results in Table 6 shows that the *hipec* approach is suitable in reducing the competitions among applications, even if the Mach LRU-like policy match the access patterns of applications. The video data is accessed in the sequential read-only pattern that each page frame will be referenced only once. Thus, the video player can reuse the recently accessed page frames and does not need to request more free page frames from the system. Consequently, the number of the flushing operations will be reduced because the system needs not to reclaim pages from the database generator. Most of the performance gain in Table 6 is obtained from reducing the number of flushing dirty pages from the database generator. There are 4% to 13% improvements for the video server in the *hipec* environment over the Mach environment, even with much less physical memory. Many real world applications have the similar sequential access pattern, such as compiler, linker and Latex word processing applications that can benefit from *hipec*. Figure 6 shows the normalized results of Table 4, 5, 6 by assuming that the elapsed time evaluated under the Mach kernel is 100%.

	Allocated Page Frame Size			
	64MBs	48MBs	32MBs	16MBs
	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%

Table 4: Elapsed time for the nested join operator running under the Mach and the privileged *hipec* environments.

5.3 Performance evaluations of multiple applications

In this section, the applications, presented in the previous section, run simultaneously under both of the Mach and the unprivileged *hipec* environments to show that the *hipec* mechanism not only can speed up single application, but also improve the overall system performance. The co-running applications are: **J.**) the nested join operator with 60 Mega bytes outer relation and 4 K bytes inner relation. **T.**) the binary tree index access query for randomly accessing 100000 tuples. **M.**) the sequential MPEG video player playing 129.58 Mega bytes MPEG video

Access Size		Allocated Page Frame Size			
		64MBs	48MBs	32MBs	16MBs
		Elapsed Time (in seconds)			
100	original	12.00	12.43	11.98	19.95
	hipec	12.07	12.13	12.18	17.47
	ratio	100.58%	97.59%	101.67%	87.57%
1000	original	52.94	54.05	61.67	82.75
	hipec	51.33	56.11	61.88	81.01
	ratio	96.96%	103.81%	100.34%	97.99%
10000	original	283.80	319.49	385.52	621.71
	hipec	279.29	321.39	384.27	616.53
	ratio	98.41%	100.59%	99.68%	99.17%
100000	original	1656.75	2344.04	3393.79	6008.17
	hipec	1621.13	2299.00	3328.06	5917.32
	ratio	97.85%	98.08%	98.06%	98.49%
1000000	original	15279.53	22338.70	33497.12	60313.42
	hipec	13953.89	19997.67	28907.42	51296.06
	ratio	91.32%	89.52%	86.30%	85.05%

Table 5: Elapsed time for the tree indexed data access query running under the Mach and the privileged *hipec* environments.

	Allocated Page Frame Size			
	64MBs	48MBs	32MBs	16MBs
	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%

Table 6: Elapsed time for mpeg_play+database generator running under the Mach and the privileged *hipec* environments.

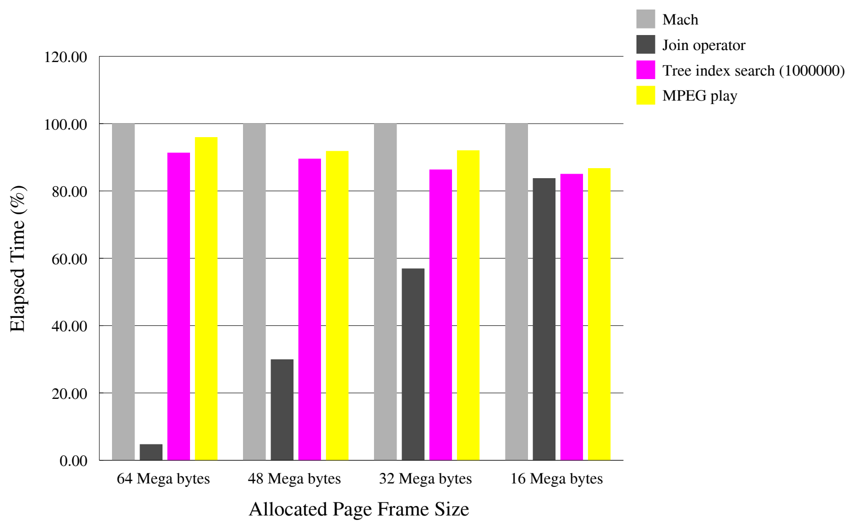


Figure 6: Normalized elapsed time for single application running under the Mach and the privileged *hipec* environments.

data with 31.16 Kbytes/sec access rate. **G.**) the database generator sequentially generating a 60 Mega bytes database. When running under the unprivileged *hipec* environment, the nested join operator implements the MRU-like policy for the outer relation. The tree index access query operator implements the prioritized *FIFO*2 policy for the memory mapped index file and the data file. The MPEG video player implements the FIFO policy, while the database generator is left without modification. There are 7 combinations for evaluation. Table 7 lists the evaluating results.

Analysis

Since all the applications are unprivileged, the page frames are competed among the applications. More page faults and page replacements occur when compared to the previous evaluated results that only single application runs under the privileged *hipec* mechanism. However, when running under the unprivileged *hipec* environment, the system performance is improved from 7% to 40% in different application combinations than under the Mach environment. Figure 7 shows the normalized result by assuming the elapsed time measured under the Mach kernel

AP combination		Allocated Page Frame Size			
		64MBs	48MBs	32MBs	16MBs
		Elapsed Time (in minute)			
J+M	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%
T+M	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+T+M	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%
T+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+T+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+T+M+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%

Table 7: Elapsed time for co-running multiple applications under the Mach and the unprivileged *hipec* environments. All the values are the averages from 5 independent evaluations.

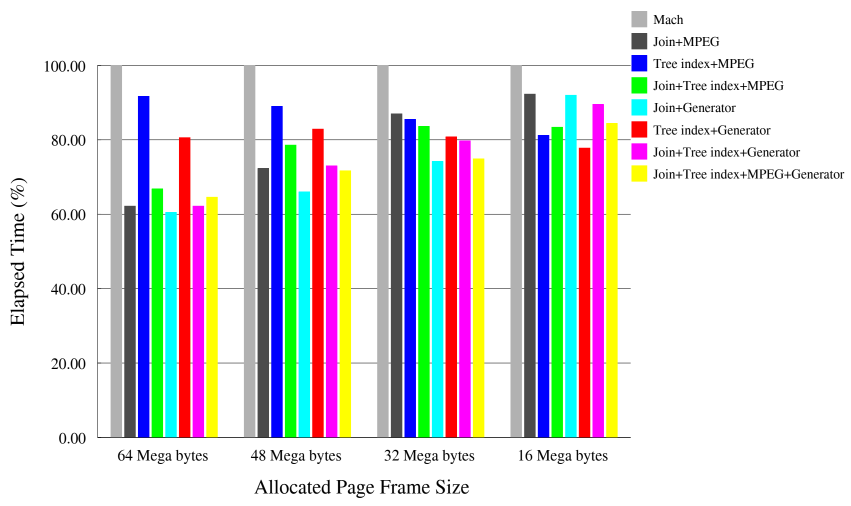


Figure 7: Normalized elapsed time for co-running multiple applications under the Mach and the unprivileged *hipec* environments.

is 100%.

6 RELATED WORK

Many research prototypes and developing systems have addressed memory caching problems. Mach [1] exports the external memory management (EMM) to user applications. EMM interface is powerful in moving data between the storage device and the application virtual address, but it lacks interfaces for applications to handle page replacement policy for the memory-mapped data. McNamee's PREMO [18] extends the EMM interface to export the page replacement facilities to applications. The kernel-maintained information[11] can be obtained by invoking their system calls. Sechrest's POD [22] extends the Mach system further in exporting the physical memory management to user applications. An application-specific PageOut Daemon (POD) is created to handle the physical memory management for external memory object. As the the kernel data structures are shared between the kernel and the user applications, which poses a potential safety problem, only trusted applications can invoke the exported interface. Neither PREMO nor POD have addressed system performance and resource management issues, such as the page frame reclamation policy. Both implementations use the Mach IPC mechanism to export the service which creates 10% to 14% overhead in their evaluations.

Spring [12] has an external paging interface similar to Mach except it separating the caching object from the pager object. The caching objects are controlled by the kernel without any participation of user applications. V++ [7] uses the segment manager (SM) to handle page faults and has interfaces to request and migrate page frames to and from different segment managers. It uses a memory market(MM) approach [6] to allocate page frames among segment managers. All the operations and requests involve transferring control among different address spaces that huge IPC communication overhead is expected. In addition, the MM

[11]In their implementation, only the referenced and modified bits (R&M) can be retrieved

approach is complex in calculating the I/O cost and needs to be well tuned when the system configuration is changed. The system performance is not addressed either, though the application access pattern knowledge can be used to improve the system performance. The recent Cache Kernel [8] argues the flexibility of existing micro kernel operating systems. User applications can have their own specific *Application Kernel* (AK) to meet their specific requirements. The Cache kernel and the *Application kernel* communicate via the memory-mapping IPC in loading and unloading the kernel object descriptors. Context switching in the AK is time-consuming, as it will call IPC functions many times and need to load and unload many kernel object descriptors. Their design is adequate for embedded systems , but not general purpose, memory-intensive applications.

Other developing systems, the *SPIN* [3], Exokernel [11] and VINO kernel [23], have the ability to export the memory caching management to user applications. Specific applications can dynamic load the executable object codes into the operating system kernel to tailor the system service to match their needs. The system safety is based on the advanced compiling techniques and the software based fault isolation techniques [25]. Though their approach will create least overhead in extending the system service, the compiling software techniques cannot be trusted in detecting the dynamic misbehaved operations. To fetch a page frame from an empty queue is one of the example. Dynamic capability-based mechanism can be used to protect the kernel resource from misbehaved accesses. However, new overhead will be also created by adding the capability-based safety mechanism. The mechanism to prevent infinite policy execution is not addressed in their designs. The infinite policy execution is hard to be detected at the compiling time by existing software techniques.

Cao has introduced a two-level file cache management policy [4] to employ the application access pattern information to increase the file system performance that the results are shown in [5]. However, their design omits the communication techniques among objects in the system, and, instead, modifies the file system directly. Previous researches in network packet demultiplexing [19, 16, 26] have also used the policy interpretation approach. Applications can program their filter in the filter language, and the system interprets the filter policy and forwards the packet to its destination. The popularity in Packet Filter implementations also shows that the policy interpretation approach is simple to use, flexible enough and little overhead.

7 CONCLUDING REMARKS

Though many memory-intensive applications have designed their own buffer management, these applications are still suffered from the unmatched kernel-controlled memory caching management [24]. By extending the page frame management policies to user applications, the applications can have their own memory caching management to match their access patterns. The application and the system performance can be increased due to the application-specific memory caching management.

The *hipec* project aims at both extending the page frame management policy to user applications and increasing the system throughput by integrating the application information to global page frame management. Applications can program their specific page frame management policy in *hipec* command set and load them into the kernel. When any page fault or page replacement operation happens, the operating system interprets the commands and performs the corresponding memory management operations to match the access patterns of user applications. The *hipec* mechanism is efficient without expensive domain-crossing and context switching overhead. In addition, the system safety is not compromised since no object codes of applications can be inserted into the kernel. The *hipec* command set itself can be treated as a portable interface and its command set can be extended to meet future needs. Additional mechanisms are implemented for guaranteeing the system safety, such as policy interpretation timeout detection, dynamic and syntax check of *hipec* commands.

Through the empirical evaluations, the *hipec* is proved to be efficient, and can increase the performance of applications up to many folds and the overall system throughput. The experience, concept, design and implementation of the *hipec* project would benefit the research in the research of the field of application-specific resource management.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, 'Mach: A New Kernel Foundation for UNIX Development', *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, USA, July 1986, pp. 93-112.

[2] 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).

[3] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, E. G. Sirer, '*SPIN* - An Extensible Microkernel for Application-specific Operating System Services', *Technical Report 94-03-03*, University of Washington, 1994.

[4] P. Cao, E. W. Felten and K. Li, 'Application-Controlled File Caching Policies', *Proceedings of the USENIX SUMMER 1994 Technical Conference*, Boston, Massachusetts, USA, June 1994, pp. 171-182.

[5] P. Cao, E. W. Felten and K. Li, 'Implementation and Performance of Application-Controlled File Caching', *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, California, USA, November 1994, pp. 165-178.

[6] D. R. Cheriton and K. Harty, 'A Market Approach to Operating System Memory Allocation', *Technical Report*, Stanford University, 1992.

[7] K. Harty and D. R. Cheriton, 'Application-Controlled Physical Memory using External Page-Cache Management', *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, USA, October 1992, pp. 187-197.

[8] D. R. Cheriton and K. J. Duda, 'A Caching Model of Operating System Kernel Functionality', *Proceedings of the First Symposium on Operating Systems Design and Implementation,* Monterey, California, USA, November 1994, pp. 179-194.

[9] H. T. Chou and D. J. DeWitt, 'An Evaluation of Buffer Management Strategies for Relational DataBase Systems', *Proceedings of the 11th International Conference on Very Large Data Bases,* Stockholm, August 1985, pp. 127-141.

[10] R. P. Draves, 'Page Replacement and Reference Bit Emulation in Mach', *Proceedings of the USENIX Mach Symposium*, Monterey California, USA, November 1991, pp. 201-212.

[11] D. R. Engler, M. F. Kaashoek, J. W. O'Toole Jr., 'The Operating System Kernel as a Secure Programmable Machine', *ACM Operating Systems Review*, 29, (1), 78-82 (1995).

[12] Y. A. Khalidi and M. N. Nelson, 'A Flexible External Paging Interface', *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, San Diego, California, USA, September 1993, pp. 127-140.

[13] K. Krueger, D. Loftesness, A. Vahdat and T. Anderson, 'Tools for the Development of Application-Specific Virtual Memory Management', *Proceedings of the ACM Eigth Annual Conference On Object-Oriented Programming Systems, Languages, and Application*, Washnigton, DC, USA, September 1993, pp. 48-64.

[14] P. C. H. Lee, M. C. Chen and R. C. Chang, '*hipec*: High Performance External Virtual Memory Caching', *Proceedings of the First Symposium on*

Operating Systems Design and Implementation, Monterey, California, USA, November 1994, pp. 153-164.

[15] P. C. H. Lee, M. C. Chen and R. C. Chang, 'HiPEC User Manual 1.0', *Technical Report CIS_95_06_04*, National Chiao Tung University, Taiwan, 1995.

[16] S. McCanne and V. Jacobson, 'The BSD Packet Filter: A New Architecture for User-level Packet Capture', *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, USA, January 1993, pp. 259-269.

[17] J. D. McDonald, 'Particle Simulation in a Multiprocessor Environment', *Proceedings of AIAA 26th Thermophysics Conference*, XX YY, June 1991, pp. aa-bb.

[18] D. McNamee and K. Armstrong, 'Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies', *Proceedings of the First USENIX Mach Workshop*, Burlington, Vermont, USA, October 1990, pp. 17-29.

[19] J. C. Mogul, R. F. Rashid and M. J. Accetta, 'The Packet Filter: An Efficient Mechanism for User-level Network Code', *Proceedings of th 11th ACM Symposium on Operating Systems Principles*, Austin, Texas, USA, November 1987, pp. 39-51.

[20] D. Rotem and J. L. Zhao, 'Buffer Management for Video Database Systems', *Proceedings of the Eleventh Internalional Conference on Data Engineering*, Taipei, Taiwan, ROC, March, 1995, pp. 439-448.

[21] J. V. Sciver and R. F. Rashid, 'Zone Garbage Collection', *Proceedings of the First USENIX Mach Workshop*, Burlington, Vermont, USA, October, 1990, pp. 1-15.

[22] S. Sechrest and Y. Park, 'User-Level Physical Memory Management for Mach', *Proceedings of the USENIX Mach Symposium*, Monterey Californial, USA, November 1991, pp. 189-200.

[23] M. Seltzer and Y. Endo and C. Small and K. A. Smith, 'An Introduction to the Architecture of the VINO Kernel', *Technical Report TR-34-94*, Harvard University, 1994.

[24] M. Stonebraker, 'Operating System Support for Database Management', *Communications of the ACM*, 24, (7), 412-418 (1981).

[25] R. Wahbe and S. Lucco and T. E. Anderson and S. L. Graham, 'Efficient Software-Based Fault Isolation', *Proceedings of the 14th ACM Symposium on Operating System Principles*, Asheviile, North Carolina, USA, November 1993, pp. 203-216.

[26] M. Yuhara and B. N. Bershad, 'Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages', *Proceedings of the 1994 Winter USENIX Conference*, San Francisco, California, USA, January 1994, pp. 153-163.