HiPEC: High Performance External Virtual Memory Caching

Chao-Hsien Lee[†], Meng Chang Chen[‡] and Ruei-Chuan Chang[†‡]

Department of Computer and Information Science[†]
National Chiao Tung University, Taiwan, ROC
paul@os.nctu.edu.tw

Institute of Information Science[‡]
Academia Sinica, Taiwan, ROC
{mcc,rc}@iis.sinica.edu.tw

Abstract

Traditional operating systems use a fixed LRU-like page replacement policy and centralized frame pool that cannot properly serve all types of memory access patterns of various applications. As a result, many memory-intensive applications, such as databases, multimedia applications and scientific simulators, induce excessive page faults and page replacement when running on top of existing operating systems.

This paper presents a High Performance External virtual memory Caching mechanism (HiPEC) to provide applications with their own specific page replacement management. The user specific policy, programmed in the HiPEC command set, is stored in user address space. When a page fault occurs, the kernel fetches and interprets the corresponding policy commands to perform the user-specific page replacement management. Experimental results show that HiPEC induces little overhead and can significantly improve performance for memory-intensive applications.

1 Introduction

Though technological advances have greatly improved the speed and enlarged the memory capacity of computer systems, because of the increasing size of applications, it is still impossible to load all applications and their data sets into physical memory at one time. Existing virtual memory management schemes can be used to compensate for limited memory size by sharing the physical frame pool among all applications. In current operating systems, a fixed LRU-like page replacement policy is usually used to handle memory sharing. This fixed LRU-like page replacement policy performs well when the applications have limited memory requirements or random memory access patterns, but it is unsuitable for many memory-intensive applications, such as databases [27], multimedia applications [24], and scientific simulators [23].

The reasons for this are the following. First, a fixed LRU-like page replacement policy and centralized frame pool that cannot properly serve all types of memory access patterns of various applications. As a result, memory intensive applications tend to induce excessive page faults and page replacement. Since page replacement usually involves disk I/O operations that are far slower than processor computation and memory access, the performance of memory-intensive applications degenerates.

Second, the operating system kernel cannot predict application access patterns and user applications know nothing about their virtual memory caching status. Since all the applications share the same centralized frame pool, the lack of information sharing between the kernel and user applications leads to unnecessary paging activities. Ideally, if the kernel and user applications share information in page replacement decision-making, and each application manages its private frame pool, the system can achieve high performance virtual memory caching by reducing unnecessary page replacement. However, the information sharing induces expensive overhead if the kernel should transfer control to user applications, or user applications should transfer control to kernel.

In this paper, we present a new mechanism, HiPEC (High Performance External virtual memory Caching), to support application-controlled virtual memory page replacement management. HiPEC is based on the Mach 3.0 kernel but can easily be ported to other operating systems. Recent research has addressed similar virtual memory caching problems. This research will be reviewed in Section 2. The motivation for HiPEC and system design are described in Section 3. The overall architecture of HiPEC and the implementation are presented in Section 4. In Section 5, several measurements and experiments are used to evaluate the overhead of HiPEC and the performance improvement

The research described in this paper was partially supported by NSC under grant NSC83-0408-E-001-008.

for specific applications that using the HiPEC mechanism. Section 6 concludes the paper and presents suggestions for future work.

2 Related Work

Many advanced operating systems and research prototypes have addressed the virtual memory caching problems. Mach [1], V++ [8] [11], Spring [18], and SPIN [6] all put the external memory management into their designs. In Mach, an external pager [32] is responsible for paging in and paging out memory-mapped data, which can be shared in a distributed environment because each data object is represented as a Mach IPC port. External pager is powerful but it lacks interfaces for applications to handle page replacement management.

McNamee's PREMO extends the external pager interfaces so that the page replacement facility is exported to applications [21]. The system-maintained information, such as reference and modify bits for each page frame, can be obtained by invoking PREMO-created system calls. This simple and direct modification of Mach reduces the number of page faults by 15% in a synthetic benchmark program. However, PREMO does not take into account the interference from other applications. PREMO puts all the page frames in one pool, which makes specific applications susceptible to unnecessary paging activities because of interference between applications.

In addition, although PREMO provides reference and modify information, it does not supply other information, such as the number of physical frames under control, which is essential to the performance of specific applications. Moreover, the IPC overhead for communication between the kernel and external pager is high. Even if the communications are implemented by upcall, as Krueger suggested [17], it is still expensive to upcall from the kernel to the user application and then call back from the user to the kernel because of the runtime stack changes.

In Sechrest's work [28], the centralized frame pool is partitioned into separate private frame lists when new memory objects are created. Specific applications have their own PageOut Daemon (POD) to handle their own memory object management, while non-specific applications are handled by a default POD. The weakness of this approach is its lack of security: information is shared between the kernel and applications without protection. The strategy of [28] is to trust on specific application designers.

Spring [18] has an external paging mechanism similar to Mach except that it separates the memory object from the pager object. The caching objects are also controlled by the kernel without user participation. V++ and SPIN are designed for application-controlled external page-cache management. V++ uses a segment manager to handle page faults and has interfaces to request and migrate page frames to and from different segment managers. It uses a memory market approach [10] to handle global memory allocation among segment managers. However, all the operations or requests involve transferring control among different address spaces. This incurs extra IPC overhead compared to an in-kernel integrated implementation.

SPIN is an extensible operating system for dynamic creation of system services. The dynamically created objects, called *spindles*, allow applications to have specific control of their allocated system resources, such as processors, memory, and network protocols. Using optimized compiling and dynamic linking skills, SPIN provides applications with full control of allocated system resources. Applications running under SPIN can achieve maximum performance without overhead from crossing the kernel/user boundary.

HiPEC is similar to SPIN in that it does not need to cross the kernel/user boundary when executing a user-specific page replacement policy. SPIN creates its application-specific control by linking the compiled object code into the kernel. It requires dynamic compiling and linking when new services are created. On the contrary, HiPEC does not create any object codes. Instead, HiPEC interprets the specific control codes placed in the user buffer area. This design provides more flexibility and requires less modification of the operating system kernel.

Mogul's work [22], the *packet filter*, is worth mentioned, although it is not related to user level memory management. The packet filter is a kernel-resident, protocol independent packet demultiplexer. Users can program their filters in the filter language, which is similar to, but simpler than, HiPEC commands. The filter is interpreted by the packet filter when system receives a packet. The goals of the packet filter are the reduction of the rate of context switches, and easy to port and test communication protocols.

3 Motivation and System Design

Due to the variety of memory access patterns of applications, operating systems must be flexible enough to support different page replacement strategies to meet the individual needs. When several page replacement strategies run at a time, it is important to reduce the interference from each other to maintain the performance of applications. One solution is to partition the centralized frame pool into separate lists that each list is allocated to a specific application* and managed

*A *specific application* is defined as an application uses HiPEC mechanism in this paper.

by the application. In addition, the kernel and specific applications need to cooperate with each other to make good allocation and replacement decisions.

There are several communication techniques available between kernel and applications. When an application needs information from kernel, it uses system calls or sends messages to communicate with kernel. Upcalls are often used by kernel to activate applications functions. Since requiring context switches, all the techniques are expensive. Another approach is to use shared memory to map shared data structures between kernel and user applications. Though this approach can speed up data access, the data has to be collected and mapped into fixed locations which is expensive too. Moreover, if the shared area is mapped with read/write permission, the security of the operating system kernel might be compromised. Consequently, kernel crossing is the source of overhead and potential problems.

Instead of finding an efficient technique for crossing the kernel boundary, HiPEC employs an integrated in-kernel implementation. In HiPEC, a specific application only needs to place its specific page replacement policy, coded as a sequence of commands in HiPEC command set, in the user space and store the pointer in an object known to the kernel. When page replacement is needed, the kernel fetches, decodes and interprets the command codes to perform the application specific page replacement policy.

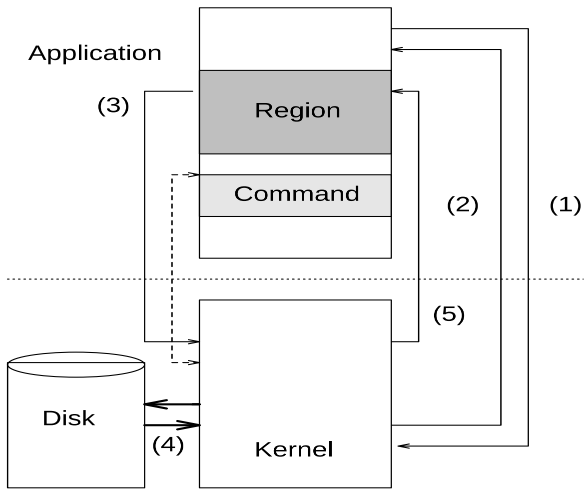


Figure 1: The Proposed Mechanism for Application Specific Control.

Figure 1 illustrates the proposed mechanism for application-specific control. The address space of specific application is partitioned into regions of continuous virtual memory. The region is the basic unit of specific control. The command codes implementing the application page replacement policy are stored in the command buffer. To activate HiPEC mechanism, the specific application first (1) calls the HiPEC-created system call with parameters including the starting address and size of the virtual memory region, and pointer to the command buffer. Normally, the specific application obtains the corresponding private frame list from the kernel, as in (2). When a page fault is generated inside the region, the application traps to the kernel as shown in the step (3). If a page replacement decision is needed, the kernel fetches the commands from the command buffer, decodes the commands and performs the required operations. After allocating physical frames for the faulted region, the kernel reads data from the disk and stores the data to the faulted address, as in step (4). Finally, in the step (5), the page fault is resolved and the application resumes its work.

This design has several advantages:

- A high degree of efficiency can be achieved because no need to cross the kernel boundary and the overhead for fetching and decoding commands is low.
- System security is guaranteed because the kernel data structure is accessed by the kernel-provided operations only. Applications cannot access protected information.
- The command codes can be treated as a portable interface for specific applications. The details of virtual memory management and system-maintained data structures are shielded from applications and designers. As a result, specific application designers do not need to consider tedious operating system internals in their design.
- High performance gain is obtainable if specific application designers know the access patterns of their applications and are able to program an efficient replacement policy in HiPEC command set.

4 HiPEC Implementation

HiPEC has been implemented on OSF/1 MK 5.0.2 operating system that extends the external memory management (EMM) interface of Mach kernel to support external virtual memory caching management. With this extension, applications can control the paging activities of memory-mapped data via the external pager [31] interface and handle the page replacement policy of a virtual memory region. Wang's implementation [30] shows that little performance overhead is incurred for running an EMM interface on BSD UNIX. This result implies that HiPEC can be ported to operating systems to support virtual memory caching management no matter whether there is an EMM interface embedded in the operating systems.

Though the current HiPEC virtual memory caching management is based on the Mach EMM interface, the concept and implementation of HiPEC is independent from the interface. Specific applications can use HiPEC to control dynamically created virtual memory regions without the help of an external pager. HiPEC has several constituents, including the *security checker, policy executor, command buffer, global frame manager, user-level pseudo code translator,* and *HiPEC command set.*

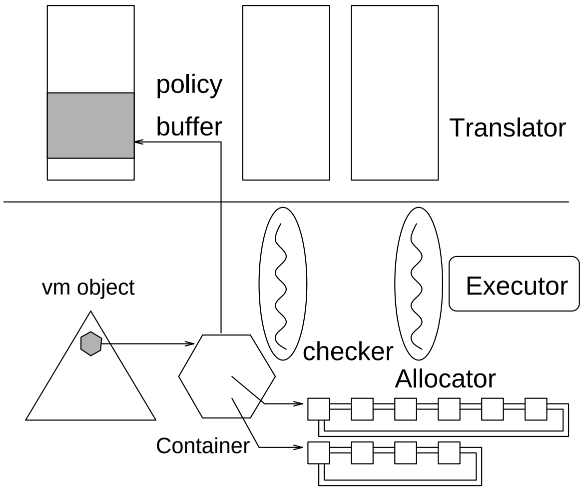


Figure 2: HiPEC Architecture.

4.1 Architecture Overview

HiPEC is composed of a set of kernel data structures, procedures, kernel threads, and user-level libraries and utilities. The architecture of HiPEC is illustrated in Figure 2. The global frame manager, implemented from Mach pageout daemon, is responsible for allocating free frames to and deallocating frames from applications. The VM object is used in the Mach kernel to represent a segment of virtual memory region that can be a memory-mapped data file or a segment of address space with the same protection attributes. One new kernel object, *container*, is added to record useful information for the HiPEC mechanism. Container is created from the zone system [29] and mounted under VM object when HiPEC is invoked by specific applications. A list of free frames, allocated by the global frame manager, is mounted under the container. The important information stored in the container includes pointer to next container, pointers to related VM objects and threads, pointers to the HiPEC command buffers, pointers to allocated free frame lists, operand array, and a timeout flag.

The command buffer is a wired down user-level area, used to store the application-specific page replacement policy. The buffer is set as read-only after a specific application invokes HiPEC and the buffer is passed as a parameter to kernel. If applications attempt to modify the contents of the policy buffer, a write fault occurs. The page fault handling routines check the address of the fault, and terminate the application with a error message.

The policy executor, called by the page fault handler, fetches the HiPEC commands for the event, decodes them, and performs the operations. Since executor resides in kernel address space, it can fetch the commands without kernel crossing or stack changing. The overhead introduced is just the time for fetch and decode several HiPEC commands.

The security checker is implemented as a kernel thread to check illegitimate HiPEC commands and detect abnormal policy execution. In the current implementation, the checker checks whether HiPEC commands have an invalid format or are inconsistent. The checker is periodically awakened to detect timeouts of policy executions. Since policy execution is performed in kernel mode, bad policies from malicious users or due to program mistakes can compromise system integrity and degenerate performance. The security checker ensures the robustness of the system. The detailed structure of each component of HiPEC is discussed in the following subsections.

4.2 HiPEC Commands

0	8	16	24
Command Code	Operand	Operand	Flag

Figure 3: The HiPEC Command Format.

A HiPEC command is a 32-bit long word that contains an 8-bit operator code and up to three operands, as depicted in Figure 3. The operand is an 8-bit long integer which is used as an index to one entry in the operand array. The operand array is stored in a container with up to 256 entries. Each entry in the operand array is a pointer to a variable. The types of the variable can be as simple as an unsigned integer, or as complex as the virtual memory page structure or page queue list.

There are 20 commands in the current implementation: *Return, Arith, Comp, Logic, EmptyQ, InQ, Jump, DeQueue, EnQueue, Request, Release, Flush, Set, Ref, Mod, Find, Activate, FIFO, LRU* and *MRU.* The syntax and semantics of each command are listed in Table 1.

Each command is implemented in the kernel as a macro or a procedure call. HiPEC commands range from complex commands, such as the page replacement policies FIFO, LRU, and MRU, to simple ones, such as Arith and Comp. The more complex a com-

Command	Binary	Op1	Op2	Flag	Operations
Return	00000000	op	—	—	The end of execution. Return value is stored in *op*.
Arith	00000001	op1	op2	flag	Arithmetic operations for integer operands.
Comp	00000010	op1	op2	flag	Comparison operations for integer operands.
Logic	00000011	op1	op2	flag	Logical operations for boolean operands.
EmptQ	00000100	op	—	—	Test if queue *op* is empty.
InQ	00000101	op1	op2	—	Test if page *op2* is in queue *op1*.
Jump	00000110	CC			Branch to next command. *CC* is the command counter.
DeQueue	00000111	op1	op2	flag	Move the page *op1* from queue *op2*.
EnQueue	00001000	op1	op2	flag	Add page *op1* to queue *op2*.
Request	00001001	Size			Request *Size* frames from frame manager.
Release	00001010	op	—	—	Release *op* frame to frame manager.
Flush	00001011	op	—	—	Flush page *op* to disk.
Set	00001100	op1	flag1	flag2	Set or reset reference or modify bit of page *op1*.
Ref	00001101	op	—	—	Test if page *op* is referenced or not.
Mod	00001110	op	—	—	Test if page *op* is modified or not.
Find	00001111	op1	op2	flag	Given virtual address *op2*, find the associated page frame *op1*.
Activate	00010000	op	—	—	Invoke another policy event. *op* is the event number.
FIFO	00010001	op	—	—	Execute FIFO page replacement policy for the *op* queue.
LRU	00010010	op	—	—	Execute LRU page replacement policy for the *op* queue.
MRU	00010011	op	—	—	Execute MRU page replacement policy for the *op* queue.

Table 1: The HiPEC Command Set.

mand is, the less overhead it creates because the policy executor does not need to fetch and interpret many commands during execution. While the simple commands induce more overhead in executing the page replacement policy, they are flexible for application designers to program a specific policy. Though only 20 commands are defined in the current implementation, they are powerful enough for many specific applications. Since the HiPEC command code is 8 bits long, there can be up to 256 different commands. It is easy to add new commands to HiPEC if more commands are needed to handle page replacement. A HiPEC program can be created by a user-level translator from a high-level pseudo code program or by hand coding.

When an event occurs, a segment of a HiPEC command program is called to handle the event. There is no limitation of the number of events that a specific application can define. However, a specific application at least has to handle the two HiPEC-defined events, **PageFault** and **ReclaimFrame**. When page faults occur, the HiPEC commands for PageFault event are interpreted to obtain free frames to handle the fault. The ReclaimFrame event happens when the system needs to retrieve physical frames from user jobs. The non-HiPEC-defined events are activated by other events, which can be viewed as procedure calls. Table 2 is a simple example that implements a FIFO with a second chance LRU-like page replacement policy.

4.3 Implementation

HiPEC mechanism is initialized by two HiPEC system calls, vm_map_hipec() and vm_allocate_hipec(), corresponding to vm_map() and vm_allocate() in Mach respectively. vm_map() maps a file into the application's address space and vm_allocate() allocates a region of unused virtual memory for dynamic or temporary data. When either of these two system calls are invoked, the kernel allocates and initializes the container, allocates free page frames from the global frame manager, and checks the validity of HiPEC commands stored in the policy buffers.

4.3.1 Pageout Daemon Serves as Global Frame Manager

In the HiPEC implementation, the pageout daemon acts as the global frame manager. It allocates free page frames to specific applications, and reclaims them when applications terminate, or when other specific applications request for page frames. Since specific applications and non-specific applications share the same global frame pool, it is important to balance the allocation of free page frames between them. The global frame manager performs four basic tasks: balance, allocation, deallocation, and I/O handling.

- *Balance.* The global frame manager is also the pageout daemon, which is responsible for page allocation and page replacement for non-specific applications when page faults occur. Since the page frame allocation should be fair for both spe-

The PageFault Event

CC	Command Code				The executed operations.
0	HiPEC Magic No				Magic number used for checking.
1	02	02	0C	01	if(_free_count > reserved_target)
2	06	00	00	05	/* else */ Jump to (CC==5)
3	07	0B	01	01	Get page from _free_queue by DeQueue.
4	00	0B	—	—	Return
5	10	02	—	—	Activate Lack_free_frame event.
6	06	00	00	03	Jump

The Lack_Free_Frame Event

CC	Command Code				The executed operations.
0	HiPEC Magic No				Magic number used for checking.
1	02	02	0A	02	if(_free_count < free_target)
2	06	00	00	0E	/* else */Jump to (14)
3	07	0B	00	01	Get page from _inactive_queue by DeQueue.
4	0D	0B	—	—	Judge if the page is referenced
5	06	00	00	09	/* else */Jump to (09)
6	08	0B	03	02	Put page to _active_queue by EnQueue
7	0C	0B	02	01	Reset the page reference bit
8	06	00	00	0E	Jump to (CC==14)
9	0E	0B	—	—	Judge if the page is modified
10	06	00	00	0C	/* else */Jump to (CC==12)
11	0B	0B	—	—	Flush page
12	08	0B	01	01	Put page to free queue by EnQueue
13	06	00	00	01	Jump to (CC==1)
14	02	06	09	02	if (inactive_count < inactive_target)
15	06	00	00	14	/* else */ Jump to (CC==20)
16	07	0B	03	01	Get page from _free_queue by DeQueue
17	0C	0B	02	01	Reset the page reference bit
18	08	0B	05	02	Put page to _inactive_queue
19	06	00	00	0E	Jump to (CC==14)
20	00	00	—	—	Return

Table 2: The FIFO with Second Chance Page Replacement Policy.

cific and non-specific applications, we define a watermark *partition_burst* to monitor the allocation. When the total pages allocated to all the specific applications exceed this watermark, the global frame manager will deallocate pages from specific applications.

Currently, partition_burst is defined as 50% of the available free page frames after the system starts up. This is under the assumption that about the same number of physical page frames are requested by specific and non-specific applications and they should have equal opportunity to be served by the global frame manager. An adaptable or dynamically adjustable partition burst will be studied in the future to investigate the impact on system performance. The effect of other frame allocation methods is also worth studying. However, it is beyond the scope of this paper.

- *Allocation.* When specific applications invoke the HiPEC mechanism, they must identify the size of their memory needs. The parameter *minFrame* is passed to the kernel to request the minimum number of page frames from the global frame manager. Specific applications will keep at least *minFrame* pages during their executions. If the *minFrame* request cannot be satisfied when HiPEC is initially invoked, an error code is returned. The specific application can either run as a non-specific application or terminate and retry later. The *minFrame* of each specific application is decided and administrated by designated privileged users who have the responsibility of system performance.

  If a specific application needs more page frames, the executor executes a *Request* command to request more pages. The global frame manager grants or rejects the request depending on the number of the remaining free page frames and the status of the requester. When the number of requested page frames is more than the available page frames, the request is rejected. The executor checks the returned code to know the status of frame allocation request. Upon request failure, the executor makes the specific page replacement policy to handle the shortage of page frames. Consequently, the HiPEC executor will not be hung indefinitely for waiting the return from the global frame manager.

- *Deallocation.* The global frame manager retrieves page frames from specific applications when their VM region is deallocated. The second situation is when global frame manager has fewer available free page frames than the *minFrame* requests from new specific applications. The last situation of reclamation is when the total pages allocated to specific application exceed the *partition_burst*. The global frame manager reclaims page frames from specific applications with more pages than their minimal request(i.e. *minFrame* pages) only.

  The reclamation of frames can be normal reclamation and forced reclamation. When HiPEC is invoked, the newly created container is added to the end of the list that links all containers. A simple policy, FAFR (First Allocated, First Reclaimed), is implemented to select the victims of reclamation. The procedure of normal reclamation is that the global frame manager follows the container list and selects the first container to release page frames until the request is satisfied. The global frame manager calls the policy executor to execute the **ReclaimFrame** event of a selected specific application to return pages to the system. This ReclaimFrame event allows specific applications to decide which pages are less important and can be released.

  The global frame manager starts forced reclamation when it cannot retrieve enough page frames from normal reclamation. Since all the allocated page frames of all specific applications are linked in the sequence of the time of allocation, the global frame manager can reclaim frame pages from the list. The reclaimed dirty pages are linked to a VM object and are flushed to disk by the global frame manager later.

- *I/O Handling.* The global frame manager also performs page flushing for specific applications. When a policy executor wants to flush a page using *Flush* command, it releases the flushed page to a VM object of the global frame manager and receives a new free page from the global frame manager. The real flushing operation is done by the global frame manager later. This design prevents the executor from having to wait for disk I/O operations. Otherwise, the executor may timeout often and terminated by the checker when waiting for the time consuming disk I/O operations.

4.3.2 Application-Specific Policy Executor

When invoked by the page fault handler or global frame manager, the policy executor fetches commands from policy buffers, decodes them, and executes the corresponding operations. The policy executed depends on the type of event that occurs for the specific VM object. At the begin of execution, the policy executor first writes a timestamp into the container to record the starting time of execution. This timestamp is checked by security checker to detect timeout of policy execution. The container also contains a CC

(Command Counter) variable that is used to record the address of the next HiPEC command to be interpreted. Since the policy executor runs in kernel mode and can directly access both kernel and user address spaces, it does not need to transfer control from kernel to user applications when fetching the commands. The executor will keep running until it reads **Return** from the policy buffers.

4.3.3 In-Kernel Security Checker

The security checker is implemented as a kernel thread that checks the validity of application-specific page replacement policies. In the current version, the security checker only checks for illegal syntax of commands, such as the wrong number or illegal type of operands. Another duty of the checker is to detect timeouts of policy executions. The checker is awakened periodically by the timer. The length of sleeping time is adjusted according to whether a timeout is detected by the checker. Every time a timeout is detected, the sleeping time for the checker is halved. If no timeout execution is detected, the time is doubled. Since normally there are very few runaway policy executions, the checker sleeps most of the time and does not create enormous overhead to degenerate system performance. The formula of the sleeping time of the checker is described in the following equation:

$$\text{WakeUp} = \begin{cases} WakeUp/2 & \text{if timeout detected} \\ WakeUp * 2 & \text{if no timeout detected} \\ 250msec & \text{if WakeUp } \leq 250 \text{ msec} \\ 8sec & \text{if WakeUp } \geq \text{ 8 sec} \end{cases}$$

When the checker is awakened, it checks the stored timestamp of each container by traversing the container link list. A policy execution is treated as a timeout if the execution time is longer than the *TimeOut* period. Currently, the length of *TimeOut* period is determined manually by a privileged user. When the checker finds an executor has run longer than the timeout period, the corresponding specific application will be terminated by the checker.

4.3.4 Pseudo Code Translator and Library

It is not convenient for a programmer to design a page replacement policy by directly using the low-level HiPEC command set. We implement a pseudo code translator to assist application designers in their programming. The translator translates C language like pseudo codes into a stream of HiPEC command codes. The translator is implemented as a stand alone program and is also incorporated into the user level library. The HiPEC event is represented as a procedure call in the pseudo code program with the **Event** type. Figure 4 shows an example of a pseudo code program

```
Event PageFault() {
if (_free_count > reserve_target)
  page = de_queue_head(_free_queue)
else begin
  Lack_free_frame()
  page = de_queue_head(_free_queue)
endif
return(page)
}
Event Lack_free_frame() { /* FIFO with 2th Chance */
while (_inactive_count < inactive_target) {
       page = de_queue_head(_active_queue)
       reset(page.reference)
       en_queue_tail(_inactive_queue)
}
while (_free_count < free_target) {
       page = de_queue_head(_inactive_queue)
       if (page.reference) begin
         en_queue_tail(_active_queue,page)
         reset(page.reference)
         end
       else begin
            if (page.dirty) begin
              flush(page)
            end
            en_queue_head(_free_queue,page)
       end
}
}
Event ReclaimFrame() { ...... }
```

Figure 4: Pseudo Code Program for FIFO with Second Chance Caching Policy.

that implements a FIFO with a second chance page replacement policy.

5 Experiments and Performance Evaluation

The advantage of HiPEC over previous techniques is that it does not need to transfer control between kernel and applications. The cost is the time for fetching and decoding HiPEC commands, execution of security checker and miscellaneous processings. In this section, three experiments are designed to measure the overhead and evaluate the performance of HiPEC mechanism. The experimental results show that HiPEC induces little overhead and can significantly improve performance for memory-intensive applications.

The first experiment presents the measurements of HiPEC mechanism that are compared with other techniques. The second experiment shows negligible overhead of HiPEC for non-specific applications. The last

Evaluation	Average Time Overhead
40 Mbytes page fault Without disk I/O operations	
Running on Mach 3.0 Kernel	4016.5 msec
Running on HiPEC mechanism	4088.6 msec
HiPEC Overhead	1.8%
40 Mbytes page fault with disk I/O operations	
Running on Mach 3.0 Kernel	82485.5 msec
Running on HiPEC mechanism	82505.6 msec
HiPEC Overhead	0.024%

Table 3: Comparison — I.

experiment show the merit of HiPEC in allowing specific applications to have great performance improvement by using their specific page replacement policy. All the experiments are performed on an Acer Altos 10000 machine, which has two Intel 486-50 CPUs and 64 Megabytes main memory. One CPU is disabled during the experiments to prevent unexpected interference.

5.1 Measurements of HiPEC Mechanism

In this experiment, we want to find out the overhead created by HiPEC and compare with other techniques that are usually used to implement application-specific page replacement management. We measure the page fault handling time for accessing 40 Megabytes virtual address space both under Mach kernel and HiPEC. To make the comparison fair, the HiPEC environment has implemented the same FIFO with a second chance page replacement policy as in Mach kernel [13] and both request 40 Megabytes for their private management. In order to distinguish the effect of disk I/O on the overall execution time, we measure the elapsed time with and without disk I/O operations separately. From Table 3, the overhead incurred by HiPEC is so small that can be compensated by as few as one or two disk page I/O operations. In the experiment 3, it is shown that a specific application with right replacement policy can reduce unnecessary page replacements.

The common techniques used to provide application specific resource management are upcall and IPC. Upcalls are implemented as procedure invocations from the kernel to user applications. The overhead is mainly in allocating area for new user stack and changing stacks. In Mach, the IPC mechanism is implemented by message passing. The time for null system call is used to describe the upcall overhead. For IPC, we measure the execution time of a null IPC.

Since HiPEC overhead is mostly determined by the

Evaluation	Average Time
Null System Call	19 μ sec
Null IPC Call	292 μ sec
Simple HiPEC page fault overhead	≅150 nsec

Table 4: Comparison — II.

programmed policy, we again use the FIFO with a second chance page replacement policy as the referenced policy. The overhead created by HiPEC mechanism in simple page fault is negligible, because it is only the time to fetch and decode *Comp*, *DeQueue*,*Return* commands. We use approximation notation to represent the simple page fault overhead for HiPEC mechanism, because the time measured is too small that can be easily affected by other system activities. It is concluded from Table 4 that HiPEC is more efficient than the upcall or IPC techniques.

5.2 The System Throughputs of Modified and Unmodified Mach Kernel

In this experiment, we want to find out the overhead of HiPEC to non-specific applications. We run a synthetic system benchmarm, AIM, on the original Mach kernel and modified HiPEC kernel to compare the overall system throughput. The AIM suite III benchmark [3] is designed to compare the system performance of various platforms and operating systems. Users can tune the workload mix by giving weights to different kind of simulated jobs to measure system throughputs.

HiPEC implementation has added checking statements to Mach kernel in the page fault handling routines to decide whether the faulted virtual address is located in the regions controlled by the specific applications. Another HiPEC implementation overhead for non-specific applications is from the security checker. The checker is awakened periodically to check if there is any timeout of policy execution. The overhead of the checker depends on the number of specific applications running in the system and the frequency of timeout detected. When only non-specific applications run in the system, the overhead created by the checker is limited.

We use three different workload mixes to evaluate the system throughput. The first is the standard workload. The second workload emphasizes on the disk usage and the third emphasizes on the memory usage. The experimental results are illustrated in Figure 5. When the number of simulated concurrent users is larger than five or six, the throughput is degraded because the users jobs are competing the system resources. From the results in Figure 5, the original

Mach Kernel and modified HiPEC kernel almost provide the same throughput under these three different workload mixes. The overhead created from HiPEC does not have obvious influence on the system performance.

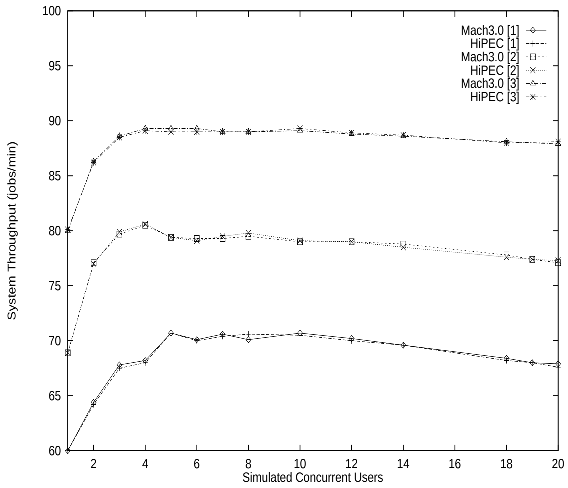


Figure 5: The Throughput on Mach Kernel and HiPEC Kernel.

5.3 Performance Evaluation of Join Operator

Join is one of the most important operations of relational database management systems. We implement a MRU page replacement policy in HiPEC for the nested-loops join operator to show the performance improvement. The other policy used is a LRU-like page replacement policy for its popularity in conventional operating systems. The inner table of the join operator is 4 K bytes and the size of outer table ranges from 20 Megabytes to 60 Megabytes. Both tables are composed of 64 bytes tuples. The output table is dumped immediately in this experiment since we want to focus on the page replacement behavior of outer table. In this experiment, the join operator is implemented as sequentially accessing the tuples in the 4 K inner table and doing join operation with every tuple in the outer table. The inner 4 K table is pinned in memory while the outer table is scanned as many times as the number of tuples in the inner table.

When the size of allocated memory is larger than the outer table, no page replacement will be needed. Otherwise, there are page replacement activities for each scan of outer table. LRU policy chooses the least recently used page frame as the page to be replaced that causes the cyclic faults for every outer loop scan. The number of page faults for LRU is

$$PF_l = \frac{OutLSize * Loop}{PageSize}$$

The *OutLSize* represents the size of the outer table. The *Loop* is the scanning times for the outer table. In our experiment, *Loop* equals to 64. *PageSize* is the physical page frame size, which is 4096 bytes for our machine.

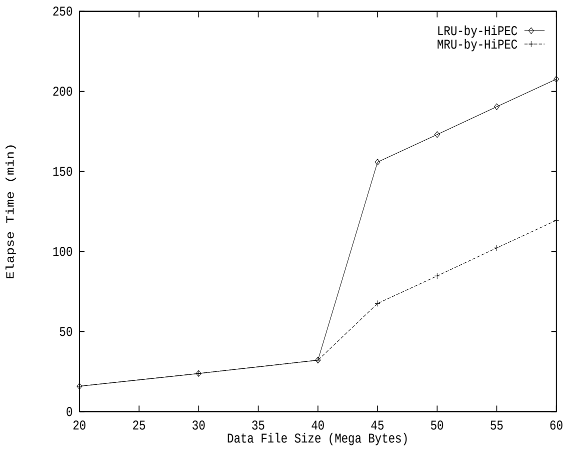
MRU chooses the most recently used page frame to be replaced. The number of page faults for first scanning the outer table is the page number of outer table. But in the successive scanning, the number of page faults is just the page number difference between the outer loop table and the HiPEC allocated memory. The total number of page faults for MRU is

$$PF_m = \frac{(OutLSize - MSize) * (Loop - 1) + OutLSize}{PageSize}$$

The variable *MSize* represents the allocated memory size, which is 40 Mega bytes in this experiment. Obviously, MRU is the right solution to the nested-loop join operation. The performance gain is

$$\begin{aligned} Gain &= (PF_l - PF_m) * PFHandleTime \\ &= \frac{(Loop - 1) * MSize}{PageSize} * PFHandleTime \end{aligned}$$

Experimental results show that a great response time gap occurs when data size is larger than available frames, i.e. 40 Megabytes. Figure 6 shows the experimental results which match the analytic result.



—	30	40	45	50	55	60
LRU	23.8	32.1	155.8	173.1	190.5	207.7
MRU	23.8	32.1	67.5	84.8	102.3	119.5

Figure 6: Elapsed Time (in min.) for The Join Operation.

6 Conclusions and Future Work

In this paper, we considered the virtual memory caching problem for specific applications. We presented the design and implementation of the HiPEC mechanism which provides efficient external page replacement management. The HiPEC mechanism does not require the kernel transfer control to the user applications when the kernel makes page replacement decisions to match the specific applications access patterns. Specific applications use HiPEC command codes to inform the kernel of their specific page replacement policies. The kernel fetches the commands, decodes them and does the corresponding operations.

The shared centralized frame pool is partitioned into private frame lists for each specific application. This separation can avoid interference from other jobs. HiPEC also implements a security checker to check the syntax of the HiPEC command sequence and detect any policy execution timeout. The security checker does not incur heavy overhead because it will sleep often if there is no frequent policy execution timeouts within the system. Several measurements and experiments are also presented to show that the HiPEC mechanism has little overhead as compared to the original integrated kernel services. Specific applications can achieve the maximum performance if the right page replacement policies are designed and managed by using HiPEC.

Though the current version of HiPEC shows successes in solving page replacement problem, there are some jobs that need more considerations in the future. First, we want to consider the page migration operations between relevant specific applications. In our current implementation, specific applications can only return the page frames to the global frame allocator. However migrating physical frames between the relevant jobs might be important and necessary. Relevant jobs can use physical frame migration to share information.

Second, we only define 20 HiPEC commands for doing application-specific control in our current implementation. They are sufficient in the current usage, but not claim they are complete. The new hardware architecture, such as flash RAM, can be managed efficiently if each specific application can control the device to meet its specific requirement. To manage these new hardware architecture, the HiPEC commands should be extended to meet the requirement. Fortunately, adding new HiPEC commands is easy in our implementation. Third, the security checker could do more than the current version in detecting malicious actions or mistakes from the specific applications. Fourth, the global frame allocation and deallocation are extremely important to the system performance. Though the current allocation policy works well, the allocation policy does not address the problems of effective resource usage and pays little attention to fairness.

Lastly, we plan to design a database management system that uses HiPEC to improve the performance and observe database requirements for future enhancement to HiPEC. This is important because the HiPEC mechanism is expected and designed for practical specific applications, not just an experimental product.

Acknowledgments

The authors would like to thank the paper shepherd, Brian Bershad, OSDI program committee as well as reviewers for their valuable comments in improving this paper.

References

[1] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanain, Jr. A., and Young, M. W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference,* July 1986.

[2] Anderson, Thomas E., Bershad, Brian N., Lazowska, Edward D. and Levy, Henry M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991.

[3] AIM Technology AIM Benchmark Suite III User's Guide. 1986.

[4] Black, D. L. Scheduling and Resource Management Techniques for Multiprocessors. Ph.D. dissertation, Carnegie Mellon University, July 1990.

[5] Bershad, Brian N., Anderson, Thomas E., Lazowska, Edward D., and Levy, Henry M. User-Level Interprocess Communication for Shared Memory Multiprocessors. In *ACM Transactions on Computer Systems,* 9(2):175-198, May 1991.

[6] Bershad, Brian N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, Emin Gün SPIN - An Extensible Microkernel for Application-specific Operating System Services. Tech. Report, University of Washington, Feburary 1994.

[7] Bolosky, William J., Fitzgerald, Robert P., Scott, Michael L. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium On Operating Systems Principles,* December 1989.

[8] Cheriton, David R. The V Distributed System. In *Communications of the ACM,* 31(3):314-333, March 1988.

[9] Cheriton, David R., Goosen, Hendrik A. and Boyle, Patrick D. Pradigm : A Highly Scalable Shared Memory Multicomputer Architecture. *IEEE Computer,* February 1991.

[10] Cheriton, David R. and Harty, Kieran A Market Approach to Operating System Memory Allocation. Tech. Report, Stanford University, CA, March 1992.

[11] Harty, Kieran and Cheriton, David R. Application-Controlled Physical Memory Using External Page-Cache Management. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1992.

[12] Date, C. J. An Introduction To Database Systems. In Addison-Wesley Systems Programming Series, Volume 1, Fifth Edition, 1990.

[13] Draves, Richard P. Page Replacement and Reference Bit Emulation in Mach. In *Proceedings of the USENIX Mach Symposium,* Monterey, CA, November 1991.

[14] Draves, Richard P., Bershad, Brian N., Rashid, Richard F. and Dean, Randall W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991.

[15] Golub, David B. and Draves, Richard P. Moving the Default Memory Manager out of the Mach Kernel. In *Proceedings of the USENIX Mach Symposium,* Monterey, CA, November 1991.

[16] Graefe, Goetz Query Evaluation Techniques for Large Database. In *ACM Computing Surveys,* June 1993.

[17] Krueger, Keith and Loftesness, David and Vahdat, Amin and Anderson, Thomas Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the 1993 OOPSLA,* 1993.

[18] Khalidi, Youself A. and Nelson, Michael N. A Flexible External Paging Interface. In *Proceedings of USENIX Association Symposium on Microkernels and Other Kernel Architectures,* 1993.

[19] Lenoski, Dean, et al. The DASH prototype: Implementation and Performance. In *Proceedings of 19th Symposium on Computer Architecture,* May 1992.

[20] McCanne, S., Jacobson, V. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference,* January 1993.

[21] McNamee, Dylan and Armstrong, Katherine Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Association Mach Workshop,* Burlington, Vermont, October 1990.

[22] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of th 11th ACM Symposium on Operating Systems Principle,* November 1987.

[23] McDonald, Jeffrey D. Particle Simulation in a Multiprocessor Environment. In *Proceedings of AIAA 26th Thermophysics Conference,* June 1991.

[24] Muller, Keith and Pasquale, Joseph A High Performance Multi-Structured File System Design. In *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991.

[25] Ritchie, D. Stuart and Neufeld, Gerald W. User Level IPC and Device management in the Raven Kernel. In *Proceedings of USENIX Association Symposium on Micro Kernels and Other Kernel Architectures,* 1993.

[26] Ruemmler, Chris and Wilkes, John An Introduction to Disk Drive Modeling. In *IEEE Computer,* March 1994.

[27] Stonebraker, Michael Operating System Support for Database Management. In *Communications of the ACM,* Vol. 24, No. 7, July 1981.

[28] Sechrest, Stuart and Park, Yoonho User-Level Physical Memory Management for Mach. In *Proceedings of the USENIX Mach Symposium,* Monterey, CA, November 1991.

[29] Sciver, James V. and Rashid, Richard F. Zone Garbage Collection. In *Proceedings of the USENIX Association Mach Workshop,* Burlington, Vermont, October 1990.

[30] Wang, Hsiao-Hsi., Lu, Pei-Ku, and Chang, Ruei-Chuan. An Implementation of an External Pager Interface on BSD UNIX. To appear in *The Journal of Systems and Software.*

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

[32] Young, Michael W. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Ph.D. dissertation, Carnegie Mellon University, November 1989.

[33] Yuhara, Masanobu and Bershad, Brian N. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter 1994 USENIX Conference,* January 1994.