GenerOS: An Asymmetric Operating System Kernel for Multi-core Systems

Qingbo Yuan, Jianbo Zhao, Mingyu Chen, Ninghui Sun
Key Laboratory of Computer System and Architecture
Institute of Compute Technology, Chinese Academy of Sciences
Beijing, China
{yuanbor, zhaojianbo, cmy, snh}@ncic.ac.cn

Abstract—**Due to complex abstractions implemented over shared data structures protected by locks, conventional symmetric multithreaded operating system kernel such as Linux is hard to achieve high scalability on the emerging multi-core architectures, which integrate more and more cores on a single die. This paper presents GenerOS - a general asymmetric operating system kernel for multi-core systems. In principal, GenerOS partitions processing cores into application core, kernel core and interrupt core, each of which is dedicated to a specified function. In implementation, we conduct a delicate modification to Linux kernel and provide the same interface as Linux kernel so that GenerOS is compatible with legacy applications. The better performance of GenerOS mainly benefits from: (1) Applications run on their own cores with minimal interrupt and kernel support; (2) Every kernel service is encapsulated into a serial process so that there will be fewer contentions than conventional symmetric kernel; (3) A slim schedule policy is used in the kernel core to support schedule between system calls with low overhead.**

Experiments with two typical workloads on 16-core AMD machine show that GenerOS behaves better than original Linux kernel when there are more processing cores (19.6% for TPC-H using oracle database management system and 42.8% for httperf using apache web server).

I. INTRODUCTION

Moore's Law indicates that the number of transistors on VLSI increases exponentially, doubling approximately every two years. Now, there are billions of transistors on a single die [9]. Chips with four cores are common while tens even hundreds of cores will appear within few years [2]. As a result, the total number of processing cores available in a shared memory multiprocessing system is very large. Designing an operating system to scale well with so many cores is a big challenge. In a conventional shared memory multiprocessing system, all of the cores are equivalent. Each core can run any task without regard to the memory location in which their data resides. This symmetric structure works well when the number of processing cores is small. However, with the increase of total core number, its performance will be limited due to the complex shared data structures and more and more contentions [8, 18]. Boyd-Wickizer et al. shows that Linux on 16 cores can have poor network performance when processing many short connections due to contention for shared network structures in the kernel [7]. Our experiments with oracle database management system and apache web server also show the lower performance of Linux when there are more cores. Another critical issue is that the reliability of such an operating system cannot be guaranteed when the core number is large. The parallel threads in the kernel are complex and each of them can access all shared kernel data structures. If any thread encounters an error, the whole kernel will crash. In addition the conventional symmetric kernel could not work in a heterogeneous multi-core system that has been becoming a trend intended to save power and improve performance. There are several studies focusing on improving the scalability of operating system on multi-core or manycore platform such as Corey [7] and McRT [22]. They both dedicate cores to handle specific functions and data, showing an efficient solution to operating system scalability on multi-core system, especially for the heterogeneous system. Corey is an exokernel-like [11] operating system with new low-level abstractions that allow library operating system and applications to control all inter-core sharing. A Corey prototype runs on 16-core AMD and Intel machines. Their measurements demonstrate that Corey can scale better than Linux. However, Corey is a totally new design that cannot support the large numbers of legacy applications which is very important for the adoption of any new operating system. McRT is a software prototype of an integrated language runtime that was designed to explore configurations of the software stack for enabling performance and scalability on large scale CMP platforms. It can achieve near linear improvements in performance and scalability for desktop workloads. But, like Corey, McRT needs modify the application to run on it.

This paper presents GenerOS, which is the acronym for 'a general operating system for multi-core'. It is an asymmetric operating system kernel works on multi-core systems and is implemented through modifying a mature operating system instead of rewriting from scratch. In this way a great deal of legacy applications could run directly in our system which is a key limitation in Corey and McRT. The work sounds very complex and tedious at first but we managed to implement it with certain techniques.

In GenerOS, processing cores are categorized as application core, kernel core and interrupt core, each of which is dedicated to a specified function. Kernel functionality is provided by separate kernel servers running on dedicated cores so that there will be fewer contentions than symmetric kernel. Applications run on their own cores with minimal interrupt and kernel support. If one of them needs some kernel service, the runtime environment in the application core will send a corresponding request to a proper kernel server. After the request is handled, the runtime will inform the application and it will resume execution. Moreover, interrupt handler could be dealt with on separate cores as well. All of these cores communicate with each other through shared memory.

A brief overview of the architecture of GenerOS is introduced first, and then experiences with the system are discussed, including implementation, experimental evaluation and analysis of the experimental results. The primary contributions of this work are (1) designing and implementing a new operating system architecture that categorizes the cores into three types: application core, kernel core, and interrupt core; (2) encapsulating kernel function as a serial process (e.g., TCP/IP server [4, 17], File System server); (3) a slim schedule policy used in the kernel core to support scheduling between system calls with almost zero overhead; (4) providing the same interface as Linux kernel in order to be compatible with legacy applications and drivers; (5) an evaluation on 16-core AMD machines that demonstrates the system can work well and has a better performance over Linux.

Throughout the process of design and implementation, to support real complex applications was one of the main goals. So we evaluate the system with commercial workloads [13] which are used in the vast majority of shared memory multiprocessing systems. We choose two benchmarks in these workloads: TPC-H using oracle database management system and httperf using apache web server. TPC-H currently represents decision support system (DSS) [3]. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. Httperf is a tool for measuring web server performance [15]. It provides a flexible facility for generating various HTTP workloads and for measuring server performance. It is fairly easy to install the above workloads in GenerOS. Moreover, all the environment and workloads are identical unless the operating system kernel. So, the comparison of the experimental results in GenerOS with the Linux kernel is more reasonable. The rest of this paper is organized as follows. Section 2 presents GenerOS architecture. Section 3 introduces its implementation, including the slim schedule policy which has less overhead in the context switch of two system calls. Section 4 evaluates GenerOS in several respects. Section 5 relates GenerOS to previous work. Section 6 concludes the paper and presents our plans for future work.

II. GenerOS Architecture

The target platform of GenerOS is shared memory multiprocessing system, such as SMP, NUMA, and heterogeneous multiprocessing system. Figure 1 shows the overall architecture of the system. It categorizes the cores in a shared memory multiprocessing system into three types: application core, kernel core and interrupt core. Each type of core has dedicated function and they communicate with each other through shared memory.

A. *Application core*

Application core can only be used by application processes. The idea for this type of core is driven by the principle that applications should not be disturbed by irrelevant events. Most of the kernel procedures and interrupts can be removed out of the application core to leave a clean execution environment for applications. This lightweight runtime environment is used to intercept system calls from applications and forward it to a remote kernel core. All of the concrete work of system calls will not execute in application cores. GenerOS reserves all of the system calls in conventional operating system. So applications need not be modified to fit in with a new system call interface.

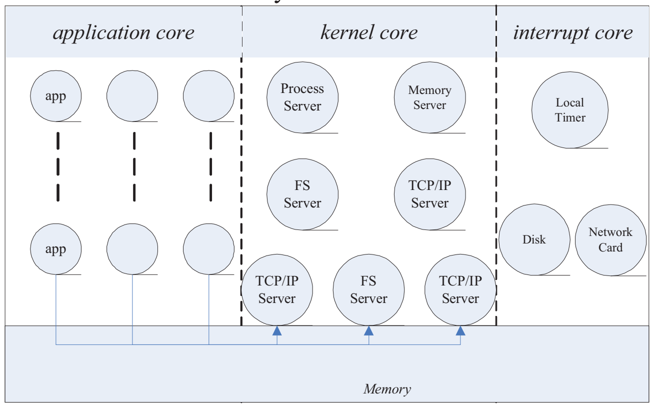

Figure 1: Overview of GenerOS

GenerOS provides a system call route table to redirect all of the conventional system calls to a proper kernel core. Once the runtime receives a system call, it will search the route table to find a proper kernel core which serves the specific function. Then, the runtime will form a request on behalf of the system call and send it to the kernel core's request queue. When the request is in queue, the application thread will be scheduled out and another ready-to-run application thread can make use of the application core. In this way, system can achieve more throughputs.

As for conventional operating system, an application and the system call invoked by it are both run in the same core. CPU must first switch to kernel mode to do system call and then switch back to user mode after accomplished it. The context switch overhead between kernel/user spaces is high. To make matter worse, the instruction and data used in kernel will pollute cache [5] which will decrease the hit ratio of the application. In contrast to this, GenerOS runtime environment is designed to be a very lightweight one. Application will spend less overhead to enter kernel mode and vice versa. Furthermore, because the execution of system call is in remote kernel core, the local cache will be influenced slightly. In such way, the performance of applications will be improved.

On the other hand, although most of interrupts can be removed from the application core such as interrupt of network adaptors or disks, it must support some absolutely necessary interrupts to deal with the unexpected events. For example, when a signal for some thread arrives, the runtime environment must hang up the current running thread and handle the situation. Moreover, there will be a lot of notifications from the kernel core or the interrupt core which will be introduced later. All of the above circumstances require application core to support basic interrupt mechanism.

B. *Kernel core*

Several kernel servers run on kernel cores such as TCP/IP server, File System Server, etc. Each of the servers is a serial process which provides specific function for applications. A kernel server always resides in kernel mode and no kernel/user switch will happen. By default, there is only one kernel server for one type of system calls. However, if some type of system calls is too many to handle, it is necessary to increase the number of such kernel server. As shown in Figure 1, there exist three TCP/IP servers and two File System servers. In this case, the balance of the same type of kernel server is very important. It is the duty of runtime environment in application core to select proper kernel server to send its system call requests.

C. *Interrupt core*

Interrupt core is used to deal with most of interrupts from network interface, disk, or local timer. In computing, an interrupt causes the processor to save its state of execution via a context switch, and begins execution of an interrupt handler. After the execution, the processor will restore the previous saved content in order to resume the interrupted task. It is evident that the performance of application will be affected by these irrelevant interrupts, especially when they are frequent. In fact, most of the interrupts need not stop the application's execution, because these interrupts are used to inform operating system the need for attention or just modify a variable in the kernel such as jiffies in Linux kernel. Therefore, most of the interrupts can be dedicated to interrupt core, leaving application core a clean execution environment.

To sum up, GenerOS classifies the functionality of cores to improve the performance of both applications and the operating system. Applications can only execute on application core, while operating system service runs on kernel core. Neither of them will influence another. When the number of processing cores is small, this method may not be suitable because the resource is limited. Sharing is the principle business for this condition. However, with the increase of the core number, sharing may not be proper anymore. In a conventional shared memory multiprocessing system, each core can enter kernel mode and execute kernel code. In this way, contentions will increase greatly as the number of processing cores increase. GenerOS addresses this problem by transforming a parallel kernel to a serial one. There may be a lot of kernel servers to which applications could send system call requests. In such a system, the performance of operating system will increase with the increase of the number of kernel servers, because each of kernel servers is a serial one and contentions between them are less. The more kernel servers present, the more system calls will be handled and the performance of applications will be improved accordingly. Of course, the number of application cores should remain unchanged; otherwise there may be a decrease in performance. Moreover, given the fixed number of processing cores, how to adjust the number of application cores, kernel cores and interrupt cores to reach the highest performance is nontrivial. Besides performance, there are at least three advantages of such system. First of all, as kernel cores provide services in the form of server daemons, it is easy to replace the old method with a new powerful one if only we keep identical with the older interface. Second, the corrupt of a kernel server will not cause the entire operating system come to ruin which takes place immediately in a conventional shared memory multiprocessing kernel. At last, GenerOS naturally supports heterogeneous architecture which is getting more popular. In a classic heterogeneous system, there may exist tens or hundreds of computing cores, but only two or four full function cores. Conventional operating systems can only be installed at the full function cores and the computing cores will not be utilized at their utmost. In contrast, GenerOS can easily be matched to the system: applications run on computing cores and some kernel servers run on full function cores.

III. GENEROS IMPLEMENTATION

There are totally 288 system calls in Linux kernel 2.6.25 for x86_64 platforms. GenerOS supports all of them with the same interface as Linux. These system calls are categorized into multiple types, each of which is assigned to a kernel core. In such a way, different types of kernel servers are generated such as File System server and Network server. We classify system calls with the interest of such a condition that a specific type of application may require only a part of the whole system calls. For example, a web server mainly need network system calls, while a database management system may require amount of file system calls. For a kind of application, specific kernel server will be turned on to handle its request. If one server is overloaded, more identical servers could be created. In current implementation of GenerOS, 6 types of system calls are introduced: file system, network, signal, IPC, process and others. This section introduces our methods to implement the system.

A. *Runtime at Application Core*

In Linux kernel 2.6.25, there exists a global system call table which consists of 288 elements for x86_64 architecture. Applications invoke some system call through the predefined system call number. We replace all of the 288 system calls in the table with our functions, but reserve all of the system call numbers and their meanings. Once an application invokes some system call, the route will directly calls for the modified function. The following simple piece of code illustrates this method:

```
const sys_call_ptr_t syscall_table [__NR_syscall_max+1]  = {
   [__NR_read] = &generos_sys_read,
   [__NR_write] = &generos_sys_write,
   ……
   [__NR_timerfd_gettime] = &generos_sys_timerfd_gettime;
};
```

After catching a system call, the runtime environment will find out which server provides the needed service, instead of executing directly on the core. There is a kernel server route table in the environment which is static at the present time. It is initialized in the boot stage of GenerOS and will not be changed during computer operation. So when some configuration cannot afford the requests from applications, it is necessary to reboot the machine and modify the parameter in the grub which will be parsed in GenerOS and result in creating proper number of kernel servers. There may be multiple identical kernel servers to provide service to applications. It is important to select a proper server to handle the application core's requests. For simplicity, GenerOS adopts a round robin method. If there are two identical kernel servers, application core will first send a request to kernel server 1 and then send the next one to kernel server 2. Even using such an easy method, the load balance of kernel servers works efficiently.

After finding out a proper kernel server, the runtime environment will pack up the system call to form a request message and then send it to the kernel server through shared memory. After the request is sent out, the application thread will be scheduled out and another ready-to-run application thread can make use of the processing core. After the system call handled by remote kernel server is completed, the application will be waked up to resume its execution.

B. *The Internal Process of Kernel Core*

The most important component of GenerOS is a variety of kernel servers running on kernel cores. As introduced previously, each of the servers is a serial procedure which is implemented as a single thread. Several kernel servers may appear in the same time. GenerOS binds a kernel server to an individual core in order to increase the parallelism among different servers. Ideally, all of the kernel servers should have no sharing data structures and then no contentions while they are running. However, since the implementation is based on Linux kernel, interactions between different kernel servers are inevitable, especially for the identical type of servers. Even so, the contention in kernel mode, the kernel/user context switch overhead and the cache miss rate are all decreased in this implementation.

In Linux, both of the application and the system call execute in the same core, while the system call executes in different processing core from the application in GenerOS. Therefore, executing environment must be rebuilt to run correctly, such as per processor data structure, page table, etc. Besides this, a proper organization must be set to deal with plenty of system call requests correctly. In our current implementation, two queues are used in each server to handle these requests. The first one is "request queue" that stores all of the system call requests from application cores. Another queue, "wait queue", is designed to store unfinished system calls that need to wait for I/O to complete. Moreover, an innovative slim schedule is designed to substitute the complex and expensive schedule procedure with a lightweight one.

1) *System Call Environments Rebuilt*

In Linux kernel 2.6.25, per-processor data structure is used by each core in the processing of a system call. There is no problem when the application and the system call run in the same core. However, in GenerOS, the concrete work of a system call is no longer executed in the same core as the application. So the per-processor data structure must be modified to fit for the environments with the system call. At least 3 values must be reset properly before the execution of a system call in the data structure: mmu_state, active_mm and pcurrent. All of them relate to the application who invokes the system call.

On the other hand, a kernel server is always running in kernel mode, looking as if page table is useless. However, as some system call requires copy data from or to user space, it is still necessary to prepare the correct page table environment; that is to say, register CR3 must be properly set. After rebuilding these environments, a system call can run correctly in kernel core.

2) *Request Queue and Wait Queue*

There may be a large amount of system call requests from application cores in a very short period of time. How to organize these unfinished requests is nontrivial. In GenerOS, two queues in each kernel server are used to handle these requests. When an application requires some system call, the runtime environment in the application core will form a request on behalf of the system call and add it to a proper kernel server's request queue. Meanwhile, all of kernel servers always keep watching on their request queues. Once a new request is inserted into the queue, a procedure will be triggered at once. In the major cases, there may be several application cores, but only one kernel core. Figure 2 gives us some idea of how it works.

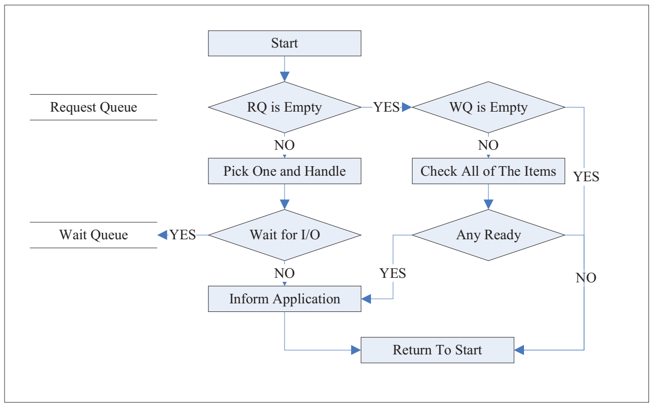

Figure 2: Request Queue and Wait Queue

At the beginning, the kernel server will examine the request queue (RQ for short in the figure) to find out whether the queue is empty or not. If not, it will pick up the first system call request and deal with it at once. In fact, there are two cases in the process of a system call. In the simpler case, the system call will be handled directly and then inform the application core by a finished message. However, some system call may require data from network adaptors or disks which always need to give up CPU. Obviously, the procedure should not be scheduled out to wait for the data because the whole process is serial. If it is scheduled out, other system calls will have no opportunity to be handled. In this case, GenerOS just put the system call request to the wait queue (WQ for short in the figure) and pick up a new request from request queue to handle if it exists. At the right moment, all of the system call requests in the wait queue will be processed and then the application will be informed.

3) *Slim Schedule*

A kernel server is a serial thread to handle plenty of some kinds of system calls from application cores. In Linux-2.6.25, a process may be scheduled out when the system call is executing in kernel mode and waiting for some resources. In GenerOS this will prevent other system calls in request queue from executing because the kernel server is scheduled out. This problem is solved by an innovative schedule method named slim schedule. When kernel server executing some system call invokes the schedule function, a process switching will not happen, but just insert the system call request to the wait queue and pick up a new one in the request queue. After the wait condition finishes, the request in the wait queue will be handled again by the kernel server. In such way, even if a system call invokes a schedule function, it will not influence the handling of other system calls.

A pair of functions are used to implement this mission: setjmp and longjmp [19]. The slim schedule method is mainly made up of three parts: code associated with the request queue, code associated with the schedule function and code associated with the wait queue. Every time a system call is issued, setjmp will store the current major registers and some addresses which will be used later by longjmp. According to the return value, several different functions will be selected to execute.

a) Request Queue Part

The pseudo code associated with the request queue is presented as follows:

```
for syscall in request queue:
    ret = setjmp(&jmp1)
    if ret == 0:
        stack = private_stack
        if syscall == sys_read:
            call sys_read()
        if syscall == sys_write:
            call sys_write()
        ……
    if ret == 1:
        continue
    if ret == 2:
        inform application
```

If the return value of setjmp is 0, the kernel server will call for an associated system call directly and if the return value is 1, nothing will be done but continue to handle another system call. Return value of 2 indicates that the system call has finished and the corresponding application should be informed and resumes execution.

b) Schedule Part

It is very common that a system call such as sys_read calls for schedule function in the course of execution because of I/O delay. As all of the system calls to be processed in the request queue are handled by the same kernel server, the invoking to conventional scheduler will lead other system calls having no opportunity to be processed. So we must also modify the schedule function to prevent this from happening. The major modification is listed below:

```
schedule()
    if need long jump:
        put the syscall to wait queue
        ret = setjmp(jmp2)
        if ret == 0:
            longjmp(jmp1,1)
        if ret == 1:
            return
```

As the normal thread also calls for the schedule function to do a context switch. It is necessary to distinguish the normal schedule calling and the slim calling. Only the slim one needs long jump. In this case, the system call request will be inserted into the wait queue and be dealt with later. Then setjmp will be called the second time to store necessary information for another longjmp. Afterwards, the procedure will jump to the former place marked by setjmp function and another system call will be handled. In such a way, the current scheduled system call is inserted to the wait queue and a new system call in the request queue is handled.

c) Wait Queue Part

After the wait event arrives, the corresponding request in the wait queue will be handled continuously.

```
for syscalls  in wait queue:
    if condition is finished:
        ret = setjmp(&jmp1)
        if ret == 0:
            longjmp(jmp2,1)
        if ret == 1:
            continue
        if ret == 2:
            inform application
```

The setjmp function is called the third time and it stores the context in the place the same as the request queue because it will not be used by the request queue again. In the first time, the procedure will jump to the scheduler and resume the previous execution. In some cases, the procedure will call for schedule again and the return value will be set to 1. Then the next system call in the wait queue will be dealt with. As for return value of 2 for both request queue part and wait queue part, the procedure will jump from the end of a system call to inform the application core that its system call request is already accomplished.

C. Binding Interrupt Handler

It has been broadly tested and verified that the normal execution of an application is always disturbed by multiple unrelated interrupts [21]. Not only the context switch time will be spent, but the cache will also be polluted by the interrupt handler. So, it is useful to bind most of the common interrupt handler to a dedicated core named interrupt core. Disk interrupt, network interrupt [23] and the local timer interrupt are the major targets of this type of core. Now, disk and network interrupts can be dedicated to interrupt cores. The local timer interrupt core is not implemented and still takes place in each of cores including the application core.

To sum up, based on Linux kernel 2.6.25, GenerOS is implemented on x86_64 multi-core platforms. The first edition of GenerOS is accomplished after several months for coding, testing and tuning. Next section will discuss the performance evaluation of GenerOS versus the conventional Linux kernel on a 16-core AMD machine.

IV. Performance Evaluation

We first test the execution time of four different system calls in both Linux kernel and GenerOS to compare the internal difference and the influence of kernel server location in GenerOS. Then two typical workloads are evaluated and thoroughly analysis is processing.

A. *Experimental Setup*

Our platform has 4 AMD Opteron 8347 quadcore processors running at 1.9GHz. Figure 3 shows the architecture of the experimental platform.

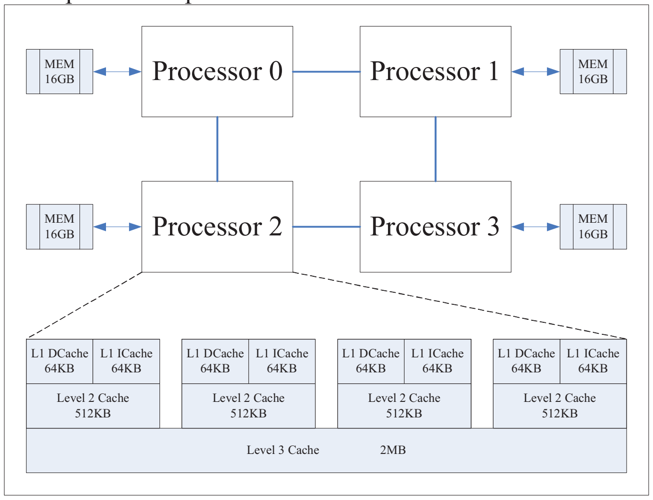

Figure 3: The Architecture of the Platform

It is obvious that not all of the processors are in the same position. For example, in order to communicate with processor1, processor0 just needs one hop. However, if it wants to communicate with processor3, two hops are needed. To deploy kernel servers, the layout must be considered first. If there are two identical kernel servers in processor1 and processor3, applications in processor0 had better choose processor1 as the kernel server instead of processor3. Furthermore, assigning two processing cores in a single processor as application core and kernel core will reduce the communication time and thus improve the performance of a system call.

B. *System Call Duration*

Although the interface of each system call in GenerOS is identical with that in Linux kernel for applications, their internal processing is quite different. As shown in Figure 4, a system call in Linux kernel consists of 3 phases, while GenerOS is made up of 5.

'ENTER' and 'EXIT' are the time spent in entering the kernel from user mode and exiting from kernel to user mode, while 'HANDLE' is the time dealing with the concrete work of a system call. These three stages are almost the same in Linux and GenerOS except that 'HANDLE' executes in a different processing core in GenerOS. As for GenerOS, two more phases are added to the whole process named 'FLYIN' and 'FLYOUT'. 'FLYIN' is the time spent in sending a system call request from runtime environment to a kernel server, while 'FLYOUT' is the reversed process. In our experiment, four different system calls are used to illustrate the difference between GenerOS and Linux: read, write, open and close.

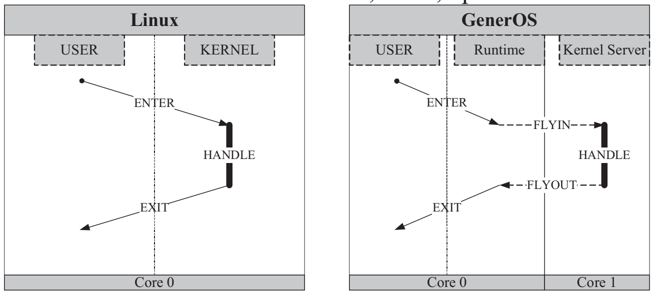

Figure 4: Stages of a System Call

1) *GenerOS versus Linux*

Based on different complexity, the four system calls can be divided into two categories: open, close and read, write. The test suite runs in the first core of processor0 in both Linux and GenerOS, while the kernel server and disk interrupt runs in other cores of processor0 in GenerOS. Figure 5 compares the time distribution of open and close in Linux and GenerOS. The left bar in both system calls is the value of Linux which consists of 3 parts from the bottom up: l-enter, l-handle and l-exit, while the right one is of GenerOS which consists g-enter, g-flyin, g-handle, g-flyout and g-exit from the bottom up. For contrast enhancement, the values of g-flyin and g-flyout are added to the top of Linux shown in dotted rectangle. It is clear that the duration of open and close in GenerOS is longer than Linux. The main reason is the overhead introduced by flyin and flyout which occupies 68.38% for open and 48.08% for close. If getting rid of these, the duration is almost the same in GenerOS and Linux. Moreover, there is a strange phenomenon that l-enter in open and close differs greatly. It is caused by the library because we record the start point of l-enter in user application which depends on library to deal with the function.

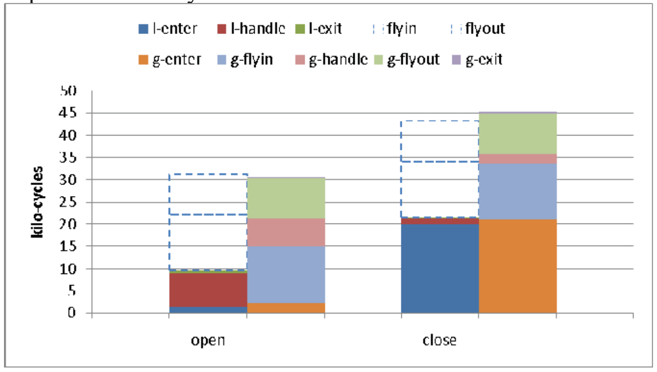

Figure 5: Open and Close System Call

On the other hand, Figure 6 compares read and write system calls which are more complicated than previous system calls (read/write 4KB data from disk). This figure is in sharp contrast to Figure 5 in which the duration of GenerOS is almost the same as Linux and the overhead of flyin and flyout is 0.13% for read and 0.46% for write.

We come to the conclusion that not all of the system calls are suitable to execute in remote kernel cores because of the overhead of 'FLYIN' and 'FLYOUT'. System calls as open and close may not be proper to execute in kernel core, while read and write are reasonable to be put in remote server.

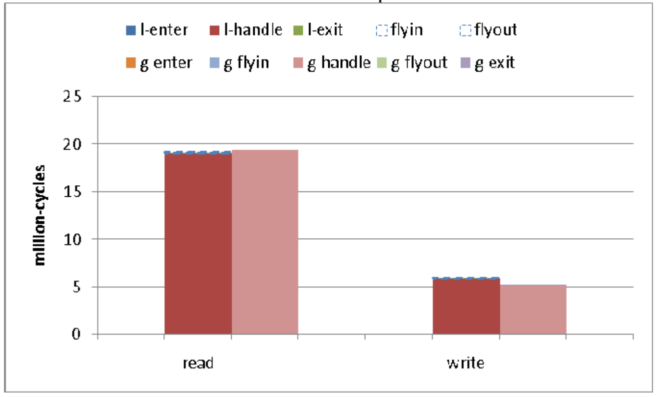

Figure 6: Read and Write System Call

2) *Kernel Server Location*

In GenerOS, the location of kernel servers is important to the performance of applications especially when there are more system call requests. If the kernel server is placed in a nearby core, the system call request will be handled more quickly and the performance will be improved. Figure 7 compares four types of kernel server locations. The four bars in each group stand for the kernel server in processor0, processor1, processor2 and processor3, respectively. In any cases, disk interrupt happens on a core near the kernel core in the same processor. As previously mentioned, the test suite which invokes system calls executes in the first core of processor0. Application core, kernel core and interrupt core will never reside in the same core. It is clear that both of the flyin and flyout value are less when send system call request to the kernel server in processor0 than that in processor3 and the values are medium when sent to processor1 and processor2. It will decrease 20.21% overhead in flyin for close system call, while 27.97% in flyout for close if selects suitable kernel server. This point should be considered in the deployment of kernel servers.

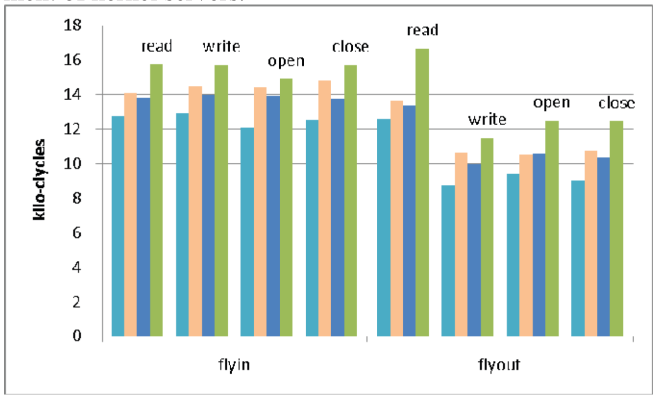

Figure 7: Kernel Server Location

The above experiments just test a single system call's behavior which can't present the whole activity of a system. In latter parts, two commercial workloads will be evaluated to give us a more thorough inspection.

C. *TPC-H Using Oracle*

This section compares the performance of GenerOS with Linux on the AMD 16-core platform using TPC-H. Oracle Database 11g Release 1 is chosen as the database management system. Figure 8 shows the result of the TPC-H power test in Linux and GenerOS using all of the 16 cores. The TPC-H Power @1G is calculated from the execution time of the 24 queries in a 1GB database, the bigger the better.

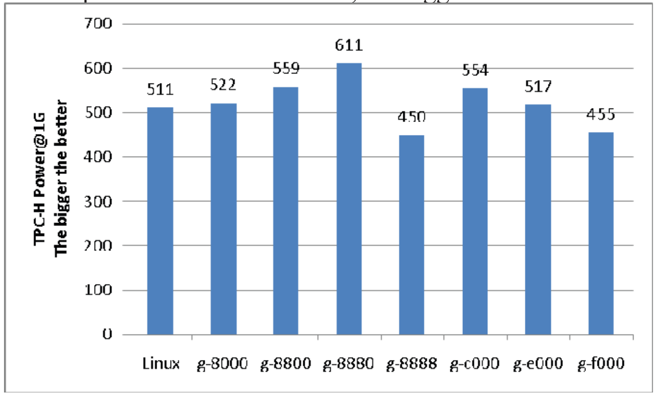

Figure 8: TPC-H Power @ 1G

Linux has only one value, while GenerOS has seven according to the layout of kernel core and interrupt core. All of the 16 cores shown in Figure 3 are numbered from core 0 to core 15. Processor0 is made up of core 0, core1, core 2 and core 3. Other processors are made up of the left cores respectively. Table 1 lists the seven configurations of GenerOS.

Table 1: GenerOS Configuration

	application core	kernel core	interrupt core
g-8000	0 ~ 14	15	15
g-8800	0 ~10, 12~14	11, 15	11, 15
g-8880	0~6, 8~10,12~14	7, 11, 15	7, 11, 15
g-8888	0~2, 4~6, 8~10, 12~14	3, 7, 11, 15	3, 7, 11, 15
g-c000	0~13	14, 15	14, 15
g-e000	0~12	13, 14, 15	13, 14, 15
g-f000	0~11	12, 13, 14, 15	12, 13, 14, 15

One important thing to note in this table is that interrupt core and kernel core use the same cores in all of the configurations. As the performance of a database mainly depends on file system calls, the kernel core is used to run file system server. Meanwhile, interrupt core is used to handle interrupts from disks. In our experiment, there is relatively small influence of disk interrupts. It is needless to waste a processing core to deal with the interrupts. So, we bind all of the disk interrupts to the kernel core.

The value of TPC-H Power@1G in Linux is about 511, while the lowest is 450 in GenerOS g-8888 configuration and the biggest is 611 in GenerOS g-8880 configuration. That is, GenerOS will increase 19.6% in the best situation. First, let's take a look at the values tested in the situation of g-8000, g-8800, g-8880 and g-8888. These four configurations have a common ground that the latter one adds an extra kernel core than previous and none of the kernel core resides in the same processor. It is clear that add a kernel core can increase some performance until four kernel cores. In this condition, we can assume that it is needless using more than three kernel cores. And because the number of application cores will degrade when adding more kernel cores, the performance of oracle database management system will decrease. Next, let's put g-8000, g-c000, g-e000 and g-f000 together to consider. This group is almost the same as the previous one except that all of the kernel cores reside in the same processor. Basically, adding more kernel cores in the same processor will not cause the increase of performance. This phenomenon can be explained with the aid of Figure 7 that all of kernel cores are far from the application cores. In this case, the overhead of flyin and flyout will be huge and the performance is decreased.

Cache hit ratio is important to the performance of an application. Higher cache hit ratio means there will be less memory traffic. Figure 9 compares data and instruction cache miss ratio in Linux kernel and g-8880 configuration of GenerOS. All of the data is collected through reading the performance counter of each processing cores [10]. The value records all of the applications' cache miss ratio during the execution of TPC-H, not just oracle processes'.

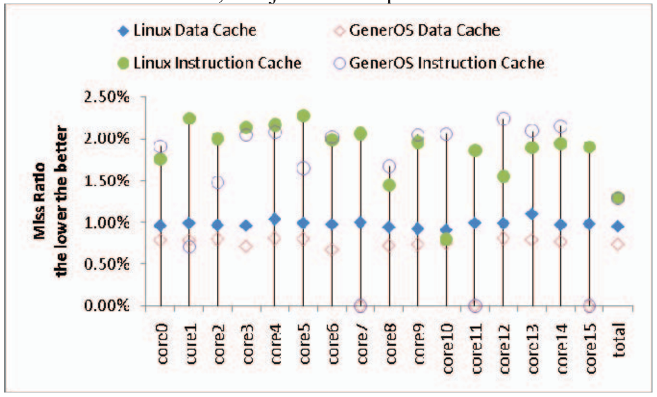

Figure 9: Cache Miss Ratio

Both of the data and instruction cache miss ratio in core7, core11 and core15 is zero in GenerOS because there are only kernel servers running in kernel mode in these cores. Each of the data cache miss ratio in GenerOS is lower than Linux in any cores, while there is not a clear trend about instruction cache miss ratio. Totally, GenerOS decreases about 21.87% (0.95% to 0.74%) in data cache miss ratio and 0.92% (1.30% to 1.28%) in instruction miss ratio.

D. *Httperf Using Apache*

Httperf is a tool for measuring web server performance. In our experiment, a popular web server apache is installed in Linux kernel and GenerOS based systems. Httperf generates various HTTP workloads in another machine and sends it to the testbed for measuring apache performance. The two machines are connected through gigabit Ethernet. Every second, httperf sends a fixed number of requests to apache and check how many replies will return, the more the better. Figure 10 shows the results from 200 requests per second to 1000 in both Linux and GenerOS. In GenerOS, processing core 15 is used as kernel core and interrupt core in the situation of g-8000-8000, while core 15 is used as kernel core and core 11 is used as interrupt core in the configuration of g-8000-800. All of the left processing cores are used as application core. When the requests are small, Linux and GenerOS can both reply almost each of requests. If the requests are more than or equal to 600 per second, a lot of requests are lost in Linux and GenerOS. The result in Linux suggests a marked reduction and the maximum value happens at 400 requests per second. While in GenerOS, the maximum value happens at 600 requests per second in the configuration of g-8000-800. Especially at 1000 requests per second, the replies are 228 in Linux versus 312 in g-8000-8000 and 326 in g-8000-800. That is, the performance in GenerOS increases 42.8% over Linux in the best situation. The trend lines of Linux, g-8000-8000 and g-8000-800 indicate that GenerOS has a better performance.

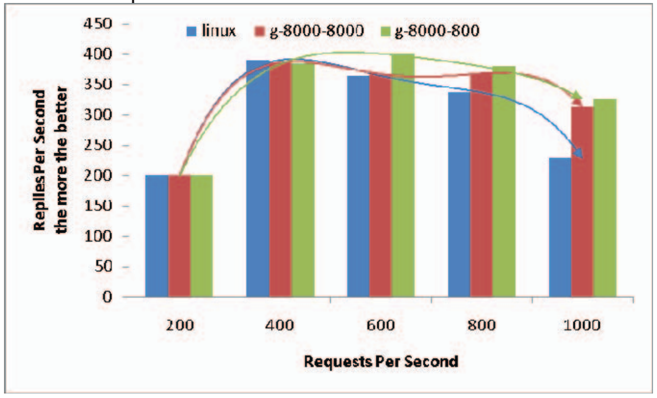

Figure 10: Httperf

Another interesting phenomenon is that the replies in g-8000-800 always exceed the value in g-8000-8000 even if the number of application core is less in the situation of g-8000-800. We may reasonably conclude that binding the network adapter interrupt to a dedicated processing core is useful for web servers.

Through the above experiments, we could see the advantage of GenerOS over Linux. Considering the dynamic configurable structure, GenerOS is suitable for a lot of situations. In our current experiments, commercial workload is the chief object of our system. As the increase of processing cores in a shared memory multiprocessing system, Linux is restricted by the complex shared data structures and more and more contentions, which is proved in our experiments. So, it is necessary to look for a good solution of this problem. In our opinion, asymmetric processing system may be a good answer.

V. Related Works

Shared memory multiprocessing platform is widely used in the server market today. There will be more processing cores in multi-core era and the scalability issue is more important than ever. A lot of works have been done to address this problem.

Mach [1, 6, 20, 24] is one of the earliest examples of a microkernel [14, 16] which revives recently in multi-core era. It provides inter process communication (IPC) functionality at the kernel level and uses it as a building block for the rest of the system. Some basic functions are implemented at kernel level ('task', 'thread', 'port', 'message'), while most other functions are at user level (File System, Device Driver, etc.). This structure is admirably suited for a multi-core platform as it is loosely coupled and light weighted. Unfortunately, the use of IPC for almost all tasks turned out to have serious performance impact. Benchmarks on 1997 hardware showed that Mach 3.0-based UNIX single-server implementations were about 50% slower than native UNIX [12].

Corey is a new operating system based on the principle that applications should control all sharing. It focuses on supporting library operating systems for multi-core applications. It is similar with GenerOS in which Corey allows library operating systems to run on dedicated cores handling specific functions and data. However, because Corey defines its own interface, the applications cannot take advantage of this system without modification. In contrast to this, GenerOS has an advantage over Corey since it is compatible with the conventional Linux kernel so that all of the applications can run on it without any change.

McRT (Many Core Run Time) is a software prototype of an integrated language runtime that was designed to explore configurations of the software stack for enabling performance and scalability on large scale CMP platforms. McRT achieves near linear improvements for desktop workloads while GenerOS focuses on the commercial workloads, such as database and web server. It is interesting that McRT can be configured to run on "bare metal" and replaces the traditional OS. Such configuration becomes increasingly attractive to leverage heterogeneous computing units seen in today's CPU-GPU configurations.

VI. Conclusions and Future Work

This paper presents an asymmetric operating system kernel for multi-core systems designed toward both performance and compatibility. Based on Linux kernel 2.6.25, GenerOS was implemented, and evaluated on a 16-core x86_64 platform. Being compatible with conventional Linux kernel, GenerOS does not need to modify, recompile, or relink applications, or libraries. Benchmarks based on oracle database management system and apache web server demonstrate that GenerOS behaves better than Linux kernel. This suggests that asymmetric operating kernel is suitable for large-scale multi-core.

However, many improvements need to be done to strengthen the performance of GenerOS in large-scale multi-core systems. The local timer interrupt should be separated from application cores and kernel cores and moved to interrupt cores to decrease the disturbance of boring interrupts. On the other hand, according to our results, different layout of kernel servers leads to different performance in fixed number of processing cores. To find the best configuration, we have to reboot the system a lot of times which is very boring. Therefore, it is necessary to tune the layout automatically given a workload and a platform. At last, the suitability of GenerOS is required to be verified on heterogeneous platforms.

Acknowledgements

This research is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 60633040, 60925009, 60803030 and the National High Technology Research and Development Program of China (863 Program) under grant numbers 2006AA01A102, 2007AA01Z115.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "MACH: A new Kernel foundation for UNIX development," Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA1986.

[2] A. Agarwal and M. Levy, "The kill rule for multicore," in DAC '07: Proceedings of the 44th annual conference on Design automation, New York, NY, USA, 2007, pp. 750--753.

[3] G. Ariav and M. J. Ginzberg, "DSS design: a systemic view of decision support," Commun. ACM, vol. 28, pp. 1045--1052, 1985.

[4] K. Banerjee, "TCP Servers: A TCP/IP Offloading Architecture for Internet Servers Using Memory-Mapped Communication," Rutgers University, Department of Computer Science, 2002.

[5] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu, "HMTT: a platform independent full-system memory trace monitoring system," in SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, New York, NY, USA, 2008, pp. 229--240.

[6] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young, "Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications," IEEE-SOFTWARE, vol. 2, pp. 65--67, July 1985.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. M. and Aleksey Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in 8th USENIX Symposium on Operating Systems Design and Implementation, 2008.

[8] S. S. K. C. Corey Gough, "Kernel Scalability—Expanding the Horizon Beyond Fine Grain Locks," in Proceedings of the Linux Symposium, 2007.

[9] I. Corporation, "World's First 2-Billion Transistor Microprocessor," http://www.intel.com/technology/architecture-silicon/2billion.htm?iid=tech_micro+2b 2009.

[10] P. J. Drongowski, "Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors," AMD whitepaper, 25 September 2008.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, New York, NY, USA, 1995, pp. 251--266.

[12] H. Hartig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schonberg, "The performance of μ-kernel-based systems," in SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles, New York, NY, USA, 1997, pp. 66--77.

[13] K. Keeton, R. Clapp, and A. Nanda, "Evaluating servers with commercial workloads," Computer, vol. 36, pp. 29-32, Feb 2003.

[14] A. Krishnamurthy, "Micro Kernels," University of Central Florida1997.

[15] H. Labs, "Httperf," http://www.hpl.hp.com/research/linux/httperf/, 2009.

[16] J. Liedtke, "On micro-kernel construction," in SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, New York, NY, USA, 1995, pp. 237--250.

[17] A. B. K. B. E. V. C. R. B. L. I. W. Z. Murali Rangarajan, "TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance," Department of Computer Science Rutgers University, Piscataway Department of Computer Science University of Maryland, College Park Department of Computer Science Rice University, HoustonMarch~28 2002.

[18] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware," Western Research Laboratory1989.

[19] R. J. Paul, J. S. Friedland, and J. E. Sagan, "Run-time error handling in FORTH using SETJMP and LNGJMP for execution control (or, GOTO in Forth)," J. FORTH Appl. Res., vol. 3, pp. 83--85, 1986.

[20] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi, "Mach: a foundation for open systems," in Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems ({WWOS}-{II}), Pacific Grove, {CA}, {USA}, September 27--29, 1989, 1989, pp. 109--113.

[21] J. Regehr and U. Duongsaa, "Preventing interrupt overload," in LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, New York, NY, USA, 2005, pp. 50--58.

[22] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale CMP environment," in EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, 2007, pp. 73--86.

[23] K. Salah, K. El-Badawi, and F. Haidari, "Performance analysis and comparison of interrupt-handling schemes in gigabit networks," Newton, MA, USA, 2007, pp. 3425--3441.

[24] A. Tevanian, D. Black, D. Golub, R. Rashid, E. Cooper, and M. Young, "MACH threads and the UNIX Kernel: The battle for control," Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA, Research paper1987.