XM-FIFO: Interdomain Communication for XtratuM

Shuwei Bai, Yiqiao Pu, Kairui She, Qingguo Zhou, Nicholas MC Guire, Lian Li
Distributed and Embedded System Lab
School of Information Science and Engineering
Lanzhou University
Tianshui South Road 222,Lanzhou,P.R.China
baishw06@lzu.cn

Abstract

A FIFO is a First In First Out data queue for sharing data between multiple tasks or threads. Fifos are one of the most basic communication mechanisms used in real-time operating systems. Especially for two real-time tasks or between a real-time task and non-real-time process, it is effective because it allows non-blocking as well as blocking semantics, which is mandatory to avert priority inversion between different domains. In the current XtratuM[7] version, there is no method available for data transfer between domains which run under XtratuMs control. So we created the XM-FIFO for the XtratuM which can be used to transfer data between real-time threads or between real-time threads and Linux user-level processes. The XM-FIFO consists of a share FIFO module, Linux FIFO device and PaRTiKle FIFO device. So far, we have released XM-FIFO V1.0, XM-FIFO V2.0 and XM-FIFO V3.0 continuously working on improvements especially at the conceptual level. Three versions adopt different technologies and provide different functions. XM-FIFO V1.0 offers the special system calls to transfer data between domains and XtratuM kernel. And the FIFOs internal synchronization is achieved by disabling interrupts - simple but undesirable in an RTOS. XM-FIFO V2.0 adopts lock-free mechanism to access the FIFO synchronously and data is copied. In the XM-FIFO V3.0, the FIFO data memory and control memory are mapped into the domain kernel space, which can increase data access speed as there is no longer an internal data copy. In the paper, besides XM-FIFOs design, the implementation of the memory mapping and lock-free mechanism in the XM-FIFO are described and comparative benchmarks interpreted.

1 Introduction

XtratuM is a nano-kernel real-time OS. It can meet the hard real-time system requirements: fast, compact, portable, simple and predictable. It supports two important device drivers: interrupt and timer. The system can be considered as a small part of the lowest operating system layers. XtratuM supports a thin software layer to interface to the domains, including virtual interrupt and virtual timer interfaces along with basic memory management functions. Linux runs on XtratuM as the root domain and manage the general purpose devices and resource except for interrupt and timer. Other domains on XtratuM are independent - that is running in there private physical address space. They have independent memory, access device I/O and so on. The independent is good for system to meet the stability and security as well as mandatory for safety related systems that should be reasonably composable. But the current XtratuM lacks the data transfer function. In order to change the status, we create XM-FIFO for the XtratuM[7].

XM-FIFO stands for the XtratuM FIFO. It is used to transfer data between domains which run on the XtratuM and can also be used within a domain (though that is not its primary intent). A FIFO is often used to connect a non-real-time process to a real-time task or to allow a real-time thread to log the message to the disk files - which is a non-RT device. Two real-time tasks also can communicate though FIFOs. FIFOs are data queues of raw bytes - no envelopes or protocol is in use. Typically, fixed-size data structures are written to FIFOs and the user

does not need to worry about the message boundary. Sometimes, the message size isnt fixed which will mandate the application to handle some sort of message end flag on its own. A FIFO is not good at big block message transfer especially if the message changed sparsely - in these cases shared memory is preferable.
XM-FIFO is bi-direction and non-blocked in the current versions. XM-FIFO V1.0, XM-FIFO V2.0 and XM-FIFO V3.0 are different in three aspects: FIFO functions offered to the users, data access mechanism and read/write synchronization. XM-FIFO follows the POSIX semantics and can be access from the user spaces(real-time user space and non-real-time user space). The user can open/close the XM-FIFO devices, read/write data from/to the files. In the XtratuM, there are sixteen XM-FIFOs with 4KB size each statically preallocated. The user cannot change the size of the XM-FIFO after compile the source code and the fixed size can be predictable for real-time system. This is also in part due to the limitations of POSIX open that does not allow passing a POSIX compliant size parameter in a reasonably compatible form.
In the XM-FIFO V1.0, there are two operations offered for the FIFO, read and write. The share XM-FIFO offers two routine symbols for the Linux kernel. As the domains are independent, we add two XtratuM system calls fifo_read/fifo_write to access the XM-FIFO from each of the domains. The XM-FIFO concurrent mechanism adopts the brute-force interrupts disable policy. Of course, the policy will make the system less responsive and impact general RTOS qualities. If the operation spend too much time, it inflicts excessive jitter on the system - thus the 4k hard-coded limit. So in the XM-FIFO V2.0 we did some changes. In the XM-FIFO V2.0, the FIFO data space is independent from the XtratuM kernel and the concurrent mechanism is changed. The lock-free mechanism is adopted which can avoid the lock-blocked concurrent and priority inversion. And the lock-free will consume less time with disable preemptability. In the XM-FIFO V3.0, the XtratuM system calls which are related with fifo are discarded entirely. The memory mapping mechanism is adopted which is the most important difference between XM-FIFO V3.0 and XM-FIFO V2.0. When the new domain is loaded, the XM-FIFOs are mapped into domain kernel space. It doesnt only map the data memory, but also control memory. The domains are independent except for the shared fifo pages.
In the background section, I will show the system call, lock-free and memory mapping mechanism respectively. A simply explanation of XtratuM and PaRTiKle[8] is in the section too. The section three shows the XM-FIFO designs and implementations of the three versions separately. How to use the XM-FIFO is presented in the section four: applications. In the section five, we will design a simple test tool for the XM-FIFO performance from XM-FIFO V1.0 to X-FIFO V3.0. The last section is conclusion of the paper. In the conclusion we will interpret what we will do in next step on the XM-FIFO project.

2 Background

Before showing what and how we have done on XM-FIFO, we should interpret some basic concepts. In the section, I will introduce PaRTiKle system, system call, lock-free and memory mapping mechanisms.

2.1 PaRTiKle[8]

PaRTiKle stands for Particular Real-Time Kernel. It is a simple real-time system kernel. The system doesnt manage any hardware and just use them. The kernel has the follow components: virtual memory management, real-time thread schedule, simple IPC between threads, virtual timer and interrupt management and so on. The PaRTiKle can run in the Linux user space, Linux kernel space and on the XtratuM as a domain independently. In our project, the PaRTiKle run on the XtratuM as the domain. The PaRTiKle system is divided into two parts as the Linux, PaRTiKle kernel and PaRTiKle user space. The threads scheduler, virtual memory management and drivers are in the kernel space. The PaRTiKle kernel can access the XtratuM kernel through XtratuM system call which we will show in the next parts. The real-time tasks or threads run in the PaRTiKle user space.
Though the PaRTiKle cannot access the hardware timer and interrupt, it has low schedule latency and interrupt latency. The values of the schedule latency and interrupt latency are 7.2us and 9.2 us respectively on the AMD 1.6G processor. The performance can meet the real-time requirement. Integrating the PaRTiKle to the XtratuM makes the system cannot support normal task running as the Linux process but also real-time tasks, running as the real-time threads. We will show how to run the PaRTiKle in the XM-FIFO usage section. And the Figure 1 shows the relation between PaRTiKle, Linux and XtratuM[1].

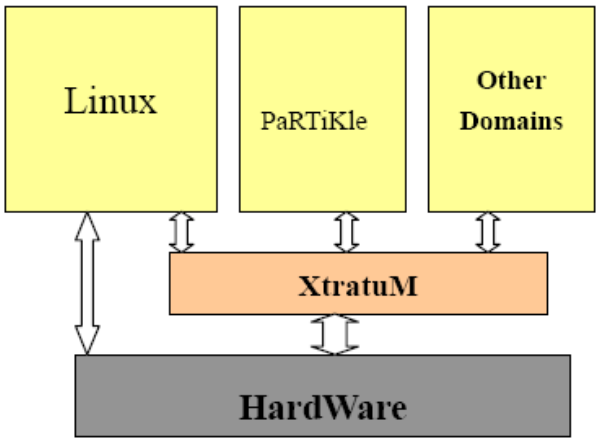

FIGURE 1: *The relation among XtratuM, PaRTiKle and Linux*

2.2 System Call

System call is one of the most important characteristics of the Linux system. The system calls split the system to two parts: kernel space and user space. The mechanism makes it easy to adopt the virtual memory mechanism and makes the system more robust. In XtratuM, the system call mechanism is adopted too. As we know, the XtratuM main module loaded by insmod tool which means the XtratuM and Linux kernel in the same address space. Another system call layer is created between in the XtratuM kernel and domains. Domains in XtratuM can only access the lowest hardware through XtratuM system calls. And the domains address is separated from the XtratuM kernel, Linux kernel and other domains. In the Linux, the system call trapped 0x80. To avoid colliding with Linux system call, the interrupt number of XtratuM system call is 0x82. In the current XtratuM version, eleven system calls are supported. In XtratuM, there is one system call table which keeps the system call number. There is a simple chart of the XtratuM call work. In the example, the call number of write_scr_sys is nine. See Figure 2 The flow chart of write_scr_sys in XtratuM .

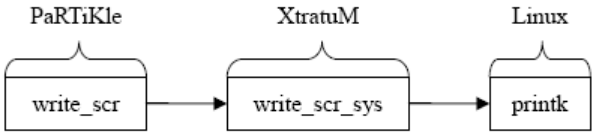

FIGURE 2: *The flow chart of write_scr_sys*

2.3 Lock-Free[2, 3, 4, 5]

There are several traditional methods to implement the synchronization such as spin-lock, semaphore, BKL, etc. But they are wait-blocked mechanism, which will cause priority inversion. So we must adopt lock-free mechanism into the FIFO access procedure such as sequence locks or RCU type mechanisms. The lock-free mechanism which can avoid the common problems associated with traditional blocked-base mechanism:

- priority inversion: occurs when a high-priority process requires a lock held by a lower priority process. The
- deadlock: can occurs if different processes attempt to lock the same set of objects in different orders;
- preemption-tolerance: if the lock holder go to sleep, and the other process want to get the lock. The result is obviously, the new process must waiting for the sleeper, which means the new process spin uselessly.[4]

In order to realize the lock-free mechanism, the basic method is CAS(Compare-and-Swap) operation. The CAS is an atomic operation with the concrete implementation depending on the computer architecture. The X86 implements the CAS operation by the**cmpxchg** command, and the command appears from X486 and later architectures in the X86 family. There are two code segments. One is the algorithm of the CAS operation with C language and another is the implementation of the CAS operation based X86 architecture.

```
int cas(int *addr,
        int old_value,
        int new_value)
{
    if(*addr == old_value) {
        *addr = new_value;
        return 1;
    }
    return 0;
}
```

```
#define CAS(adr, ov, nv) ({ \
    __typeof__(ov) ret; \
    __asm__ __volatile__( \
    "cmpxchg %3, %1" \
    :"=a"(ret), \
    "+m"(*(volatile unsigned int *)(adr)) \
    :"a"(ov),"r"(nv)); \
    ret == ov; \
    })
```

After interpreting the lock-free mechanism and CAS operation, the usage of the lock-free on XM-FIFO we will explain in Section 2.2. The XM-FIFO V2.0

discards the traditional mechanism for XM-FIFO concurrent access and it adopts the lock-free mechanism based on X86 architecture.

2.4 Memory Mapping[6]

Memory mapping is a virtual memory mechanism. Multiple processes can access the same physical memory address through different virtual memory address. In the Linux system, the user space process can access the kernel space through Linux system calls. Without the system call, the process also can access the kernel memory by the memory mapping mechanism (i.e. VSDO). As we know, the kernel memory maps the physical memory one to one (ignoring issues of more than 896MB). But sometime especially for block data transfer, we need the process space can map to the physical memory and access it directly. The mechanism will result the kernel and user process both can access the memory directly. Of course, it will be more efficiency than system call to transfer data.

In XtratuM, the physical memory is managed by the Linux memory module. When the domain loaded, the physical memory is allocated. As mentioned above, the domain are separated from each other via MMU protection which can improve the stability and security of the domains, especially for the real-time domain such as PaRTiKle. The physical memory allocated for one domain includes four types: 1) pgd table page is for keeping page tables for the domain virtual memory; 2) stack memory is allocated for the domain image; 3) the domain heap for the domain system allocated; and 4) event memory keeps the virtual interrupts and handler. The pgd page and stack in the XtratuM kernel space. The heap and event memory are mapped into domain space and used by the high level tasks, such as the event memory are used by the domain interrupt driver in the domain kernel, and the heap can be used as a memory pool for threads.

How to map kernel memory to the domain space?

Alike Linux, the XtratuM adopts two level page table mechanism, pgd and pte. The pgd table is the highest level. Every element of the pgd points to one pte page. The physical address of the pgd page is saved in the CR3 register when the relation domain scheduled. The pte element keeps the data page address. The offset of the data page is same as the virtual address offset value. There is a simple chart which showed how to convert the domain virtual address to the physical address. Of course, the physical address is converted from the kernel memory address. See Figure 3 Domain Space Address Convert and Access

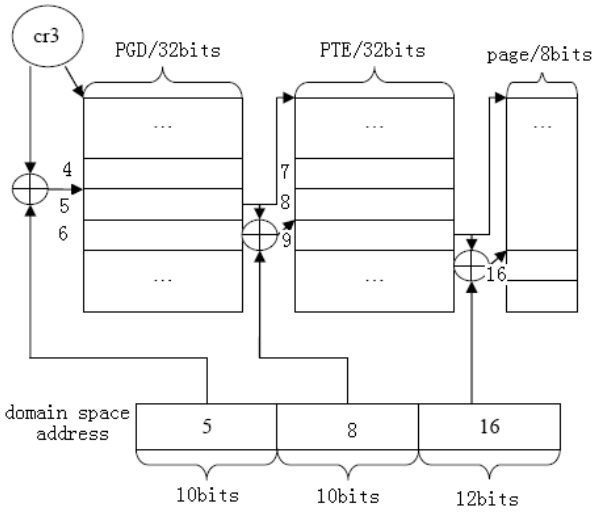

FIGURE 3: *Domain Space Address Convert and Access*

The main mechanisms which will be adopted in the XM-FIFO have been described. In the sequence section, I will interpret how to implement them in the XM-FIFO and the design of the XM-FIFO V1.0, XM-FIFO V2.0, XM-FIFO V3.0.

3 Designs and Implementations

From the Section two, you must understand the basic concepts of the main mechanisms we will adopt. If you have, it will be easy to understand the implementation of the XM-FIFOs. In the section, we also describe the design of XM-FIFOs. Some points arent shown in the above section due to space limitations. The section consists of three parts, and each part shows one version.

3.1 XM-FIFO V1.0

The XM-FIFO consists of shared fifo, Linux FIFO device driver and PaRTiKle FIFO device driver. The shared fifo is integrated into XtratuM module in XM-FIFO V1.0. Share fifo access is non-blocking, which means if the FIFOs is empty/full the read/write operation will return immediately. The FIFOs are bi-directory, each domain can read/write from/to them. The architecture of the XM-FIFO V1.0 shows in Figure 4.

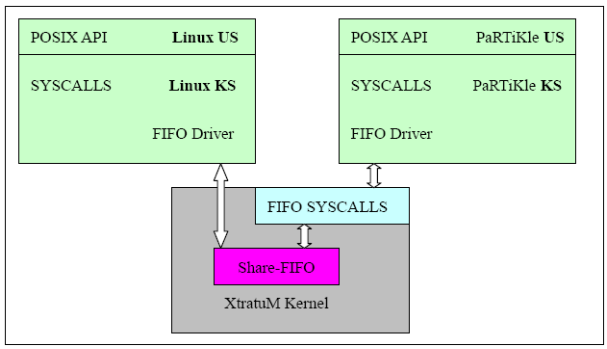

FIGURE 4: *XM-FIFO V1.0 Architecture*

From the Figure 4, its easy to see that Linux and the domains use different fifo device driver. The root domain and the normal domain access the share fifo through different interfaces. In the XM-FIFO, we run the PaRTiKle as the normal domain. In this paper, the PaRTiKle system mentioned can be thought of as the normal domain.
When the XtratuM module is loaded, the FIFO memory is statically allocated and locked into physical RAM. There are sixteen FIFOs with PAGE_SIZE bytes in each by default. The XtratuM offers two exported symbols for fifo access, xm_fifo_read() and xm_fifo_write(). Linux FIFO device driver module can call the two exported symbols to access the fifo data directly from Linux kernel context. But its different for PaRTiKle. Between PaRTiKle and XtratuM, there is a system call layer. In the XM-FIFO V1.0, we add two new system calls xm_fifo_read_sys() and xm_fifo_write_sys() to the XtratuM core. If PaRTiKle is to access the FIFO, it must call them via int 0x82 call gate. I dont think its efficient but it was the easiest to implement. The PaRTiKle user space threads and Linux user space process can access the FIFO device through the POSIX APIs, such as open(), close(), read(), write(), etc. The FIFO device names are /dev/rtf0,,/dev/rtf15 (major/minor numbers as allocated for RTLinux/GPL and RTAI). Figure 5 and Figure 6 shows the reading tree of the XM-FIFO in the Linux and PaRTiKle.

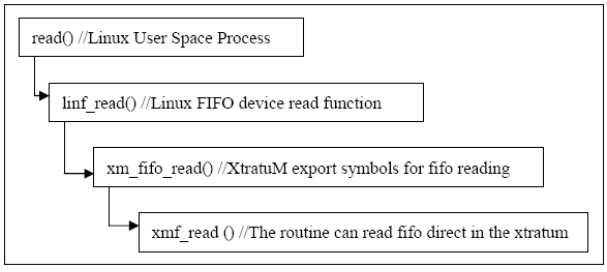

FIGURE 5: *FIFO Reading tree of Linux process*

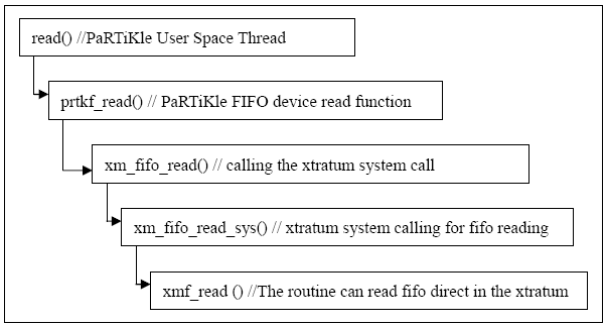

FIGURE 6: *FIFO Reading tree of PaRTiKle thread*

From Figure 5 and Figure 6, one can see that the lowest FIFO reading routine is xmf_read(), where concurrent access will happened between domains including the non-RT root-domain thus incurring potentially high delays. So we must think about synchronization in the routine to improve the contended case. In the current XtratuM version, there is no method provided to avoid race conditions except disable interrupt. We have to adopt the mechanism for handling inter-domains race condition. The algorithm of the routine is shown below.

```
int xmf_read(int index,
             char *dst,
             int size)
    begin
        ...
        hw_save_flags_and_cli(&flags);
        read_data();
        hw_restore_flags(flags);
        ...
    end
```

Race condition doesnt exit only between domains accessing XM-FIFO only, but also for the device access. In Linux and PaRTiKle, we must note the concurrent exit between processes or threads respectively. In Linux, XM-FIFO adopts read semaphore for FIFO data reading and write semaphore for write operation. In PaRTiKle, the synchronization mechanism provides less efficient semaphores only. Let me show how to use the synchronization mechanism in Linux and PaRTiKle. The algorithms are showed below.

```
int linf_read(int fd,
              char *dest,
              int size)
    begin
        ...
```

```
        down_read(&fifo.rw_semphore);
        read_data();
        up_read(&fifo_rw_semphore);
        ...
    end

int linf_write(int fd,
               const char *src,
               int size)
    begin
        ...
        down_write(&fifo.rw_semphore);
        write_data();
        up_wirte (&fifo_rw_semphore);
        ...
    end

int prtkf_read(int fd,
               char *dest,
               int size)
    begin
        ...
        sem_wait_sys (&fifo.rw_semphore);
        read_data();
        sem_post_sys(&fifo_rw_semphore);
        ...
    end
```

From the above introduction, we hope the outline of the XM-FIFO V1.0 is clear. How to use it we will demonstrate in the Section four. Now, let us look at the limitations of V1.0 of XM-FIFO.

- The interrupt disable mechanism to resolve the race condition between domains. Any other job and especially interrupts cannot be handled until the interrupts are re-enabled. This can profoundly impact the real-time behavior especially if called by a low priority domain. Obviously this mechanism would be a disaster on a multi-core system.
- The semaphore mechanism for race condition. Semaphore is block-based mechanism which will result in priority inversion. And furthermore this mechanism shows low efficiency to acquire and release the semaphores. A new synchronization mechanisms is needed.
- Use of system call to translate data between domains including root-domain. This mechanism exhibits relatively high latency which the real-time system cannot accept.
- Lack of control operations. In the XM-FIFO V1.0, there are no operations except read/write the FIFO. No controlling functions and status checking functionality is provided.

As there are some limitations in XM-FIFO 1.0, we must change or optimize those mechanisms. That is the reason we started to the XM-FIFO V2.0 and later XM-FIFO V3.0. In the next part, we describe XM-FIFO V2.0.

4 XM-FIFO V2.0

There are two differences between XM-FIFO V2.0 and XM-FIFO V1.0, lock-free mechanism replacing block-based mechanism and separating the FIFO from the XtratuM core.
In the XM-FIFO V1.0, the block-based mechanism is in use, such as semaphore and the brute-force interrupt disable mechanism. The mechanisms will decrease the system performance both with respect to average throughput as well as responsiveness. Especially in the critical area, the domain with low priority will block the highest priority domain which will cause unacceptable and unpredictable latency on the real-time domain. So in the XM-FIFO V2.0, the block-based mechanisms are discarded and lock-free mechanisms are used.
Lock-free mechanism, were outlined in section 2.3, basically building on the realize CAS operation on X86 architecture from the section. In the section, we will show the usage of the CAS in XM-FIFO V2.0.
In previous implementations of FIFOs which adopts lock-free mechanism, like the FIFO list, i.e. Michael-Scott implementation, in which nodes allocation is performed using a statically allocated set of nodes, or other implementation based dynamic nodes allocation. The static allocated implementation uses relatively large amounts of locked memory and is not suited for embedded system and dynamic implementation will potentially impact the responsiveness due to the memory allocation time. So both algorithms have disadvantages in the embedded real-time system domain. In the XM-FIFO V2.0, we use a array to realize the FIFO. As everyone knows, the array FIFO is easy to implementation. Using the array has more important reasons:

- Every FIFO need 20Bytes for saving control information which is not part of the list
- Static memory will avoid dynamic memory allocation.

In XM-FIFO V2.0, FIFOs work as circulation buffers of static size and a control area is created for each fifo. The top and bottom tags are two important members in the control area. The top is used

for write operation and the bottom tag is for read operation. The algorithms of FIFO read and write are showed below.

```
read(fd, dst, count){
  begin
    ...
    do {
        old_bottom = fifo[i].bottom;
        new_bottom = old_bottom + read_size;
        read_data();
    } while(!CAS(&fifo[i].bottom,
            old_bottom, new_bottom))
    ...
  end

    write (fd,srct, count)
  begin
    ...
    do {
        old_top = fifo[i][fifo_num].top ;
        new_top = old_top + write_size;
        write_data();
    while(!CAS(&fifo[i][fifo_num].top,
          old_top, new_top));
    ...
  end
```

From the algorithm, the CAS operation checks the top or bottom values to judge whether the read/write is preempted. But in the normal policy, the vales range is from 0 to FIFO_SIZE. So there is a high probability that the old_bottom value equals to the newest value which changed when preempted by a higher priority domain. In order to decrease the contended, the top and bottom range is from 0 to 0xFFFFFFFF, and the values are converted to a valid index of the queue before using.
There are four kinds of preemption for FIFO access, RR(read task preempt read task), RW(read task preempt write task), WR(write task preempt read task), and WW(write task preempt write task). The algorithms can avoid data inconsistent for RR, WR and RW but not WW. Figure 7 shows the RR model.

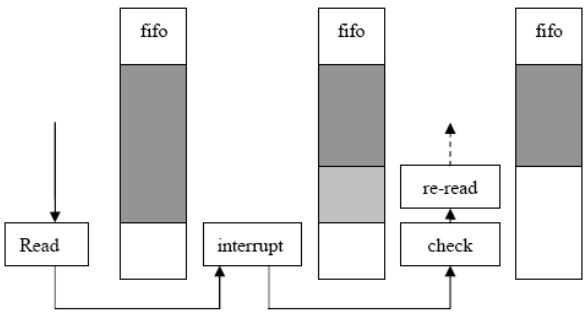

FIGURE 7: *RR Model*

The other parts of the XM-FIFO in version 2.0 are the same as XM-FIFO 1.0, which means some limitations are still present in the XM-FIFO 2.0 from 1.0. The limitations are solved in the XM-FIFO V3.0.

4.1 XM-FIFO V3.0

In XM-FIFO V3.0, the read/write XtratuM system calls are discarded. FIFOs data memory and control memory are both mapped into PaRTiKle kernel space. And more FIFO operations are supported in XM-FIFO V3.0.
As we know, the process of memory mapping is to create the page table connecting virtual and physical addresses. Let me show how the XtratuM create the page tables for the XtratuM domains. To create a page table one must prepare two things, mapping address (virtual address) and physical memory. The physical memory will be mapped into the domain space at the mapping address. As the memory access, the mapping address is converted to the pgd index and pte index firstly. Secondly to check the pgd[pgd_index] flags, if the item is occupied the flag will be set, and the pte page has been allocated for pte items or it will allocate new page to save the corresponding indexes. Thirdly, check the pte[pte_index] flags, but it is different from the pgd table check. If the flag is set, it means the virtual address has been allocated or taken up. The mapping process will be canceled. Or the physical address will be filled into the pte[pte_index].
When the domain is unloaded, the memory will be released by removing all mappings. But the system would fault because of the XM-FIFO as the physical addresses are mapped in both domains so releasing them in one could result in an invalid access from the other domain. This problem if unhandled will result in system crash. So in XM-FIFO, we add another flag mark: PAGE_SHARED which is not present in the original XtratuM memory management implementation. When the domain unloaded, the page marked with PAGE_SHARED is not physical freed until the last mapping is removed, and the item in the pte table is cleaned up. The algorithm of the FIFO memory mapping showed below.

```
allocate_fdata_page (pd,
       vaddress, index, alloc_page)
  begin
    ...// Same as the allocate_user_page()
    if [pt item of vaddress is used]
  begin
    return 0;
```

```
end
  else
  //in the allocate_user_page ()
  //allocate new physical memory.
  //In the allocate_fdata_page,
  //only calls get_fifo_data_page_addr routine
  page = get_fifo_data_page_addr(index);

  fill the pt entry table
  //set the flag value
  //_PAGE_PRESENT | _PAGE_RW | _PAGE_USER
  set the flag value
  _PAGE_PRESENT | _PAGE_RW | _PAGE_USER
  | _PAGE_ACCESSED
  //_PAGE_ACCESSED marks the page is shared.
  return address.
end;
end;
```

In the domain memory space, the physical memory is mapped from the 0x2000000. The former address is left for the device I/O. In domain space, the virtual address has been used from 0x2000000 to 0x2304000. We map the FIFO from the 0x2c00000. And the first page is mapped with the control information of the FIFOs. And the FIFO data pages are mapped in the sequence. Besides of adopting memory mapping mechanism, the FIFO control operation is added. The user can control the FIFO through ioctl() routine.

5 Usage

From the section 1 to section 3, we have showed the principle and implementation of the XM-FIFO V1.0, V2.0 and V3.0, including the difference in three versions. In this section, we will demonstrate how to use the XM-FIFO. XM-FIFOs from the V1.0 to V3.0 support a POSIX interface, so the simple process can run on all three of them. In the usage part, we will show how the Linux process and PaRTiKle communicate with each other. This section is divided three parts, Linux process, PaRTiKle thread and the limitations of the XM-FIFO usages. In the demonstration, the PaRTiKle thread send data to Linux process and the data will be printed on the screen by the Linux process.

5.1 PaRTiKle Threads

For PaRTiKle, we create a new domain in the PaRTiKle examples directory: partikle/user/examples/c_examples. If you create your task in other path, the config file needs to be changed in the Makefile which the in the current directory. Writing data to fifo is easy. You can open it as the normal device with open() and call the write routine write() to write data to the fifo. By default, there are 16 FIFO supported, from rtf0 to rtf15, with 4K size. Let me show the source code of the PaRTiKle domain.

```
$ vi fwirte.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define fifo_dev "/dev/rtf0"
int main(int argc, char *argvs[])
{
  char message[] =
       {"12345678901234567890123456789O"};
  int fd;
  int i, j, ret;
  if((fd = open(fifo_dev,
           O_RDWR, O_EXCL)) < 0) {
    printf("open device error\n");
    return 0;
   }
  for(i = 0; true; i ++) {
    ret = write(fd, message, 20);
    if(ret <= 0) break;
  }
  close(fd);
  return 0;
}
```

In the example, the task will write the message to rtf0 uninterrupted until the fifo is full. Each time 20 Bytes are written into the fifo. It should be simple to understand the code example but note that this code is not clean. In the next part I will show the Linux process source code.

5.2 Linux Processes

As in the PaRTiKle, the fifo reading program for Linux is very simple too. Firstly, the Linux fifo device driver should be loaded to provide the rtf[0..15] devices.

```
$ vi fread.c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define fifo_dev "/dev/rtf0"

int main(int argc, char *argvs[])
{
```

```
  char string[40];
  int fd, i;
  int ret;
  int total = 0;

  if((fd = open(fifo_dev,
          O_RDWR, O_EXCL)) < 0) {
    printf("open device error\n");
  }
  string[39] = '\0';
  for(i = 0; i< 10; i++) {
    ret = read(fd, string, 39);
    if(ret <= 0) break;
    printf("%s\n", string);
  }
  close(fd);
  return 0;
}
```

Loading the PaRTiKle process and then executing the Linux process will result in the data written in the context of the PaRTiKle domain to be printed in Linux user space context.

5.3 Usage Limitations

The XM-FIFOs are still not perfect now. So in this part, we will show the limitations in the XM-FIFO usage.

- In the Version 1.0 and Version 2.0, the open/close/read/write operations has been initialized, and in the Version 3.0, the flush operation which can clean up the fifo was added - necessary to ensure proper shutdown. Other operation or system call are now not available but simply the compile time settings apply (i.e. fifo size). So the operations on the FIFO are few, but enough for basic communication.
- Compile time fixed FIFO size. The size of the FIFO is 4KBs. After the XM-FIFO module is loaded, the size can't be changed. Another limitation is the number of the FIFOS which is currently 16. This limitation is not considered that serious as in general real-time systems have a-priory known and bounded resource demands.
- From XM-FIFO V2.0, we adopted the lock-free mechanism, which has not resolved the problem of more than one task writing data at the same time. So the user must avoid more than one task writing to a fifo concurrently.
- XM-FIFOs are only non-blocking. This is different than the RTLinux/GPL FIFO which will activate the rtf handler when FIFO is accessed - but this is a non-POSIX extension in RTLinux/GPL and XM-FIFOs follow strict POSIX API. A blocking semantics would allow to provide a similar mechanism as the rtf_rt_handler in RTLinux/GPL via signal (i.e. -EAGAIN).

6 Performance

From the above sections, you could see XM-FIFO versions adopt different mechanism and slightly different capabilities. So the question is what impact does this have on the FIFO performance, especially the throughput and communication time. In the section, I use the same test suit to test the XM-FIFO throughput and communication time .
There are two parameters, average throughput and minimum throughput. For every version, I test the throughput when the data blocks size are 1, 2, 4, , 4096, respectively. The test suit algorithm is shown below.

```
for(bsize = 1; bsize <= fsize; bsize *= 2)
  avgs = 0;
  mins = 0xffffff;
  ret = 0;
  for(i = 0; i < nts; i++) {
    cycles = (fsize + bsize -1)/bsize;
    rdtscll(st);
    for(j = 0; j < cycles; j++) {
      ret += write(fd, buf, bsize);
    }
    rdtscll(et);
    t = ((et - st) * 1000000 / 1602116);
    avgs += (fsize * 1000 * 1000 /t);
    mins = (mins > (fsize * 1000 * 1000/t))
           ? (fsize * 1000 * 1000/t) : mins;
    read(fd, rbuf, 4096);
  }
  avgs = avgs / nts;
  printf("%lu %16lu %20lu\n",
         bsize, mins, avgs);
}
```

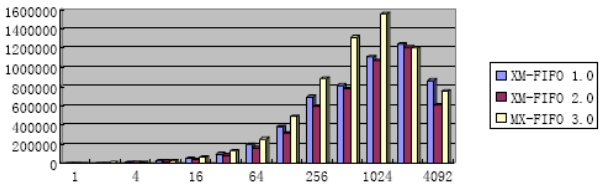

FIGURE 8: *The Minimum Throughput(K/S) of XM-FIFO*

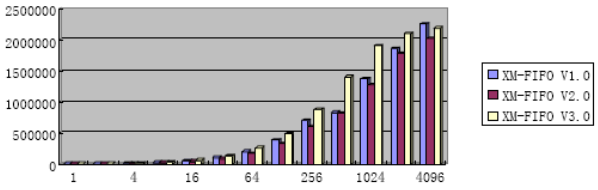

FIGURE 9: *The Average Throughput(K/S) of XM-FIFO*

From Figure[5-1], we can find the throughput of the XM-FIFO V3.0 is the highest, especially when the block size is 1024Bytes. When the block is bigger than 1024B, the throughput will decrease in the worst case. The reason is the data transfer process is interrupted. From the average throughput, we can find the throughput is increasing all the time. When the block size is bigger than 2048B, the chart is smooth. The dispersion between V1.0 and V3.0 is decreased until block size is 4096B when V1.0 throughput exceeds V3.0.
There are two value max time and min time. For each version, we test the communication time between tow domain. We also test this when the data blocks size are 1, 2, 4, , 4096. The fundamental principle is as following:

```
Heigh prority domain  low priority domain
clock_gettime
write fifo
nanosleep     ---->       read fifo
                          clock_gettime
                          nanosleep
```

Then we can get a set of communication time and get a interval which is [min, max] - the test data is as following.

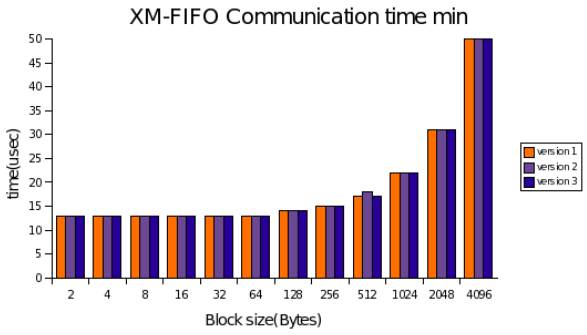

FIGURE 10: *communication min time consumption*

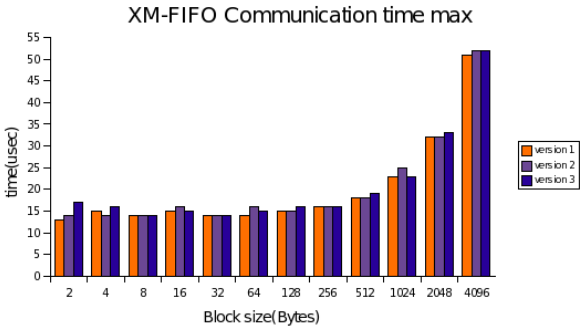

FIGURE 11: *communication max time consumption*

We can see from the result of communication time is almost the same in all three version. All of the three version has a acceptable communication time in 10 microsecond level.

Further benchmarks on highly loaded systems, especially with high priority domains/tasks running that don't use the XM-FIFO and observing the impact of low-priority tasks heavily using the XM-FIFO, on the high-priority domain is of interest. Our expectation is that the lock-free mechanism will then be of greater performance with respect to worst case latency than the locking (V1.0).

7 Conclusion

XM-FIFO can be used as a reliable real-time capable IPC mechanism between domains, which can be used to transfer data between real-time task and non-realtime task located in different domains. And XM-FIFO can connect all the domains together, which resolves the communication limitations in the early XtratuM version. The implementation of lock-free mechanism could avoid priority inversion brought by the lock-based mechanisms.
From the current performance test, we can get the transfer throughput and communication time. We must to get more information about the XM-FIFO, especially compare with other FIFO tool. So in the next step, we need to test the scheduler latency, and compare XM-FIFO parameters with other real-time FIFOs, such as RTLinux/GPL FIFO, RTAI FIFO and L4-FIFO.
Blocking semantics is in discussion as well as how to provide a better configurability of the fifo parameters within the constraints of the POSIX API. We hope to merge the current V3.0 XM-FIFO implementation into the main-stream XtratuM/PaRTiKle development trees as we believe it has reached a level

of maturity that is suitable for release to the free-software community.

8 Acknowledge

Prof. Nichols Mc. Guire given us some important technology support. XtratuM team from Universidad Politcnica de Valencia, Spain, given our some help on the PaRTiKle device driver and of course for providing a XtratuM and PaRTiKle under a truly free-software license. Last but not least DSLab offered the necessary resource and appropriate environment for the system development.

References

[1] *Introduction to XtratuM*,M. Masmano, I. Ripoll, A. Crespo,2005

[2] *An Optimistic Approach to Lock-Free FIFO Queues, Workshop on Distributed Algorithms*,Edya Ladan-Mozes, Nir Shavit,2004

[3] *Wait-free synchronization*,M.P. Herlihy, January 1991 ACM Transactions on Programming Languages and Systems

[4] *Proceedings of the 16th IEEE Real-Time Systems Symposium*,James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay, 1995, IEEE Computer Society Press

[5] *Lock-Free Techniques for Concurrent Access to Shared Objects*,Dominique Fober Yann Orlarey Stephane Letz, 2003

[6] *Linux Device Drivers, Version 3*,Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, 2005 OReilly Media, Inc

[7] *http://www.xtratum.org*

[8] *http://www.e-rtl.org/partikle/*