Journal of Theoretical and Applied Information Technology
20th April 2013. Vol. 50 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

DESIGN AND IMPLEMENTATION OF A DATA CACHE FOR UM_BUS

[1]YANG XIAOLIN, [1]ZHANG WEIGONG*
[1] Department of Information Engineering, Capital Normal University, Beijing 100048, China
E-mail: xiao_lin_yang_cnu@163.com, zwg771@yahoo.com.cn

ABSTRACT

In modern embedded systems, most of them mount with a certain amount of peripherals devices, often a large number of I / O time is consumed in the process of processor access these devices, thereby reducing the overall performance of embedded systems. However, to open up a space in the memory for caching of these device's data can resolve this problem. UM_BUS (Dynamically Reconfigurable High-speed Serial Bus) with 32 bits wide is the research object of this paper, whose bandwidth can reach 269.5M/s in the ideal condition, but the large number of I/O operations has a serious impact on bus bandwidth utilization. In order to resolve this problem, a data cache mechanism based on the structure and basic theory of cache is designed and implemented for UM_BUS in this paper. After the experimental test, the mechanism is proved to run in the UM_BUS controller driver effectively and improves the bandwidth utilization of UM_BUS significantly, therefore, enhances the performance of the bus to some extent.

Keywords: *UM_BUS, LRU, Cache, Buffer, Read-ahead*

1. INTRODUCTION

UM_BUS (Dynamic Reconfigurable High-speed Serial Bus) is a multi-channel high-speed serial system bus based on M_LVDS, which transfers data in parallel in a redundant lane with a dynamically reconfigurable method. At the same time, the UM_BUS detects the lane faults in real time and isolates the fault line, and then according to the existing line of effective, date are transmitted after dynamic restricted, thereby improving the overall system reliability and fail safety. This bus supports only one master device and 32 external devices (slave) by now. The slave device contains three address spaces: configuration space, I/O space, and storage space, in order to access these address spaces, two kinds of command are designed in the bus controller (master). One of them is short packet data frame used to access the slave device's I/O space and configuration space by master, the other is long packet data frame designed for the master to access the slave device's memory space. Every time the master accesses the slave device's memory data with the size of 1KB. In the ideal case, the bandwidth of UM_BUS with 32 lanes can be achieved 269.5M/s, unfortunately, the actual bandwidth utilization is limited as the large number of devices I/O operations in bus, which in a certain extent affects the performance of the bus. To solve this problem, drawing on the page cache mechanism in Linux [1, 2] and the drive-level caching mechanism proposed in the literature [3], we design a date cache for UM_BUS and realize it in the driver of UM_BUS controller. With this date cache, the bus controller can buffer some of data from the salve devices which is non-real time, and at the same time, device write operation is delayed. As the date cache implementation, the number of the device's I/O operations greatly reduced, therefore promoting bus bandwidth utilization and improving bus performance.

This paper is divided into five sections. The first section briefly introduces UM_BUS, as well as the problem needed to be resolved and solution method. Section 2 analyses the buffer block replacement algorithm and proposes an improved LRU algorithm realized with stack. Section 3 is the most important part of this paper, the design and implementation of data cache mechanism for UM_BUS will be discussed in this section. The test and analysis method which proves the correctness and validity of the date cache is in section 4. The last section is the conclusion of the paper and the overview of the next step work.

2. REPLACEMENT POLICY

2.1 Common replacement algorithm

The cache replacement algorithm is a major factor affecting the performance of the cache

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

mechanism, there are many kinds of replacement algorithm, such as LRU, FIFO and RANDOM are the frequently-used. In general, the advantage of FIFO and RANDOM algorithm is relatively simple to achieve, but the performance is poor, a worse problem which maybe occur when performing FIFO or RANDOM algorithm is page thrashing, that is, the cache mechanism down to nothing more than the repeated backward and forward swapping of buffer blocks[4, 5]. LRU is short for least recently used; it is based on a inference of program locality principle: the recently-used block possibly to be accessed again in future. The performance of LRU is optimal while implementation is very complex. Generally, there are two main methods to realize LRU: counter and stack. The first one usually depends on a lot of hardware support and is suitable for large-capacity cache. On the contrary, LRU with stack implementation is very simple and fit for small-capacity cache. As the slave devices of UM_BUS has a Features of small amount of data and a small-capacity cache is enough to UM_BUS, so stack method is adopted to implement LRU in this paper. Besides, some improvements are made in the algorithm to make it more applicable to the cache of UM_BUS.

2.2 The improved LRU replacement algorithm

There are two doubly linked lists, called the active list and the inactive list, are taken to realize the improved LRU [6]. Every buffer block must and only be grouped into one list, taking BLK_active status bit to mark it. The active list tends to include the buffer blocks that have been accessed recently, while the inactive list tends to include the blocks that have not been accessed for some time. Clearly, buffer blocks should be stolen from the inactive list [4, 5]. The relationship of these two lists is illustrated clearly in Fig.1. The active list is used to implement the LRU with the traditional stack method, that is, every accessed buffer block will be move to the head of list (Fig.1 ⑤). With the passage of time, this results in a kind of "equilibrium" in which frequently used buffer blocks are at the beginning of the list and least used buffer blocks are right at the end. On the other hand, the inactive list is managed with simple NRU algorithm, which not needs to move element of linked list when a buffer block is accessed. NRU is an approximation algorithm of LRU, whose implementation requires an access bit called BLK_reference [6] to mark one buffer block whether accessed or not recently. There are two functions in BLK_reference; one function is worked as the evidence which is needed when a buffer block is moved from inactive list to active list. As long as every hitting or replacement of a buffer block in inactive list is occur, a judgment of BLK_referenced should be taken. If the BLK_reference is 1, the relevant buffer block will be put into active list (Fig.1 ④), else set BLK_referece to 1(Fig.1 ②). In other words, only recently accessed buffer block will be added to the active list. It required a second proof to convert a buffer block from inactive to active; the other function is used as a mark by NRU. When one buffer block needs to be replaced, found a buffer block from the header of inactive list at first, if BLK_reference of the found buffer block is 0, then replaced it (Fig.1 ①), else set the BLK_reference bit to 0 (Fig.1 ③) and then retain the buffer block in cache and continue to check the next block. If the BLK_reference is still 1 until the last block of inactive list is checked, then the search procedure should back to the header of inactive list and continue until a block is found with BLK_reference is 0. In the actual implement, of course, it must be take into considered that moved part of buffer block from active list to inactive list when the amount of buffer block in active list is excessive after the search failure (Fig.1 ⑥).

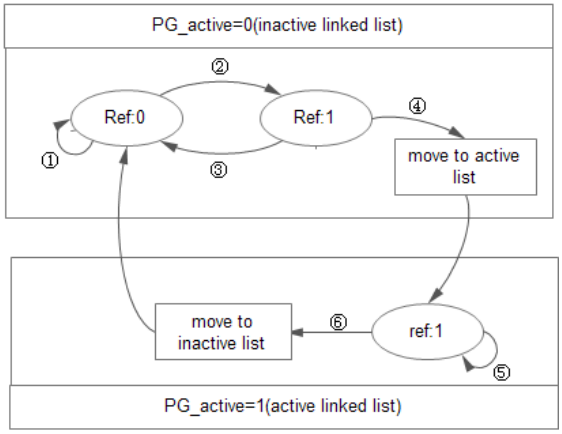

Fig1. Block movement between the LRU lists

3. DESIGN AND IMPLEMENTATION OF CACHE

The cache mentioned in this paper refers to a software-defined data structure, which is used to buffer the data got from the slave devices by master, rather than a hardware cache for accelerating memory access. It is implemented in the driver of UM_BUS controller (master). The Detail design and implementation of this cache are described as below.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3.1 Cache workflow

With the data cache, the process of master device from received to completed an I/O request is approximately shown in Figure 2.

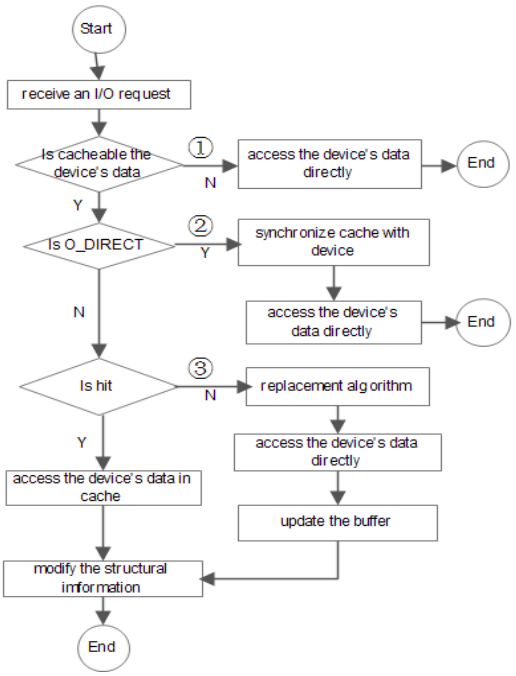

Fig2. Cache workflow diagram

When received an I/O request, the master will access the slave device with three means. The first is directly access the slave device. For real-time performance consideration, some slave devices can't buffer the device's data, only directly perform I/O operations (Fig.2 ①); Accessing the data of slave devices from cache is the second way, it is as the same as the traditional way to cache. At first searching the cache with the offset address of request data, otherwise, the replacement algorithm will be called to find a block to buffer the read-write data (Fig.2 ③); The third way will be taken when the I/O request is marked with a flag that accessing the device directly and the device's data is cacheable. In this case, the first step is to synchronize the corresponding dirty blocks in cache with the device and set these blocks to an outdated state. If the I/O request is write, it is also essential to set the buffered blocks related the device to an outdated state. At last, directly I/O operations on devices will be performed (Fig.2 ②). Among these three methods, only the first one doesn't modify the structural information of cache, the others will change.

3.2 Module division

In order to make the implementation of cache more feasible and easily, it is necessary to modularize the cache mechanism. In this paper, the cache mechanism is divided into four modules included data structure management module, I/O management module, find and replacement module and dirty blocks write-back module. Among these modules, the I/O management module can be further divided into I/O transfer module, error handling module and read-ahead module; Find and replacement module is composed with find sub-module and replacement sub-module. The relationship of these four modules is shown in Figure 3.

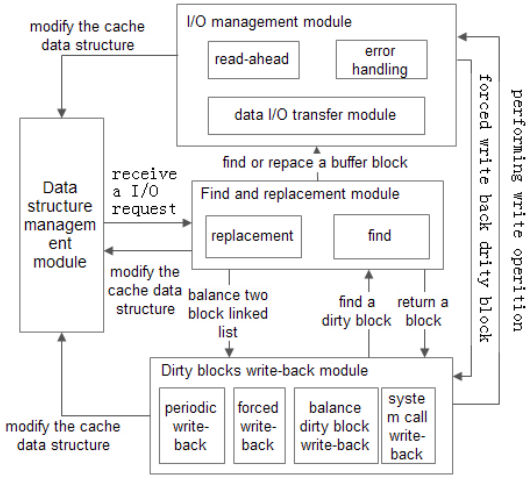

Fig3. Module relationship diagram

The basic functions of these modules will be described as below.

3.2.1 Data structure management module

There are four kinds of data structure that are designed to manage the cache information, is they are, umbus_cache, dev_inode, block and buffer. We will generally describe the basic information of each structure in this section.

① The structure of umbus_cache manages the global information of cache, including the linked list for managing device nodes and two block linked list used to implement LRU algorithm and so on. Specific members of this structure are shown in Table 1. It is noted that background_thresh and dirty_thresh fields are designed to control dirty blocks write-back. As soon as the number of dirty blocks exceeds the value of dirty_thresh, a forced write-back thread will be called and be pended until the number under the value of background_thresh. The polling Timer

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

member is a timer which is used for waking the periodic write-back thread. (Note: There are only list important members in the following table. The structure of list_head is a double-linked list. The connection between these list_head members involved in tables is shown in Figure 3).

TABLE1: THE STRUCTURE OF UMBUS_CACHE

Type	Field	Description
struct dev_inode **	*dev_tree*	Manage all of pointer to the device node address.
struct list_head	*active_list*	The head of active double-linked list
struct list_head	*inactive_list*	The head of inactive double-linked list
struct list_head	*private_list*	The head of a device with dirty data.
long	*background_thresh*	The dirty background threshold.
long	*dirty_thresh*	The maximum number of dirty blocks allowed.
long	*nrblks*	The number of block in cache.
KTIMER	*pollingTimer*	The timer for waking periodic write-back thread.
…	…	Other information such as lock.

②The dev_inode structure describes the basic information of a specified device and the linked list of corresponding blocks. Table 2 illustrates the detail members of dev_inode. There is a rw_lock number which is a Reader-Writer lock with priority for writers, it allows shared read the buffer data for multiple readers but restricts only one write operation at a time; The io_blk and dirty_blk member manage the dirty block list of the device with together. The io_blk list links dirty blocks which are ready to write back to device. These field is designed to avoid several processes flush a dirty block to device at the same time, when a process issued an request to flush a dirty block detects a write-back operation on the dirty block is performing, it will directly give up the write-back request, which is conducive to reduce I/O operations; The tags field is used to record the status of buffer blocks related to the device, including three states: TAG_EXIST, TAG_DIRTY and TAG_WRITEBACK. With the tags field, a process can quickly detect the status of a block, which greatly improves search efficiency.

③ The block structure remains the basic information of buffer blocks, and also manages a linked list of buffer structure. The specific configuration is shown in Table 3. The buffer_list linked list describes the offset address of changed data in the block; The VirtualAddress is the address of the buffer block data in memory; The Flags field records the status of the block (see Table 4).

TABLE2. THE STRUCTURE OF DEV_INODE

Type	Field	Description
Struct umbus_cache*	*UM_cache*	The pointer points to the address of cache space.
unsigned long	*devNum*	The device number
unsigned long	*flags*	The status of device
struct rw_struct	*rw_lock*	Reader-writer lock for device
struct list_head	*dev_list_dirty*	The list links the next device node with dirty data.
struct list_head	*io_blk*	The head of list which links the dirty blocks reading to write back.
struct list_head	*dirty_blk*	The head of list which links the dirty blocks
struct list_head	*blk_list_lru*	The head of list which links the blocks.
long	*tags[][]*	the state of buffer blocks
long	*i_size*	The size of device's data space.
long	*nrblks*	The number of buffer block.
…	…	Other information such as lock.

TABLE3. THE STRUCTURE OF BLOCK

Type	Field	Description
struct dev_inode*	*pdev_inode*	The pointer point to the belonging device structure.
unsigned long	*index*	The block offset in device
unsigned long	*flags*	The information of block attributes.
struct list_head	*blk_list*	The list links to the next buffer block of the same device.
struct list_head	*dirty_io_list*	The list links to the next dirty buffer block of the same device.
struct list_head	*buffer_list*	The head of the list which links buffer.
struct list_head	*lru*	The list links to the next block that is belong to either active list or inactive list.
void *	*virtualAddress*	The address of data in memory.
…	…	Other information such as lock.

③ The buffer structure records the location information of dirty data in a buffer block. When only a small amount of data in a block is modified, write-back the whole block data will waste a lot of I/O time flushing the data which is unchanged to device. However, according to the location information recorded in the buffer structure, we can

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

be able to write back fewer dirty data to device, which is benefit to improve the bandwidth utilization of UM_BUS. Table 5 shows the detail members of this structure.

TABLE4. FLAGS DESCRIBING THE STATUS OF A BLOCK

Flag name	Meaning
BLK_locked	The block is locked, it is involved in a device I/O operation.
BLK_new	The block is initialized.
BLK_uptodate	The data of the block is update and invalid.
BLK_error	Exception information thrown in a device I/O operation.
BLK_referenced	The block has been recently accessed.
BLK_active	The block is in the active block list.
BLK_dirty	The block has been modified
BLK_private	The buffer list in block is not NULL.
BLK_writeback	The block is writing back.

TABLE5. THE STRUCTURE OF BUFFER

Type	Field	Description
struct block *	*b_blk*	The pointer point to the belonging block.
unsigned long	*from*	The start offset address of dirty data in block.
unsigned long	*To*	The end offset address of dirty data in block.
struct list_head	*b_this_blk*	The list links the next buffer structure.

The relationship of these four structures is shown in Figure 4.

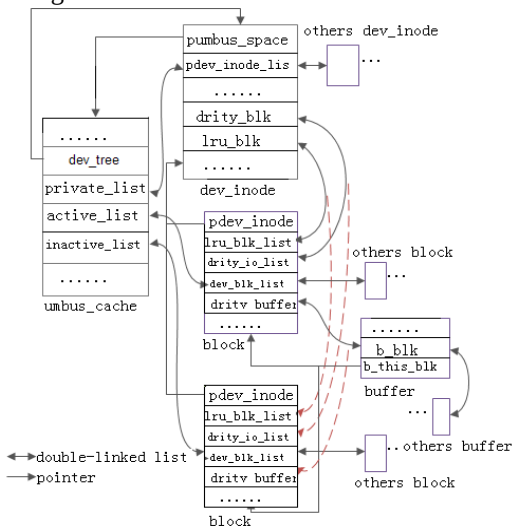

Fig4.The relationship of five data structures diagram

3.2.2 I/O management module

This module mainly manages the slave device's access operations. There are two ways to access the device's data, including read/write the device's data with cache and direct I/O transfer the device's data bypass cache. Just as the Figure 3 shown, the module can be divided into three sub-modules: I/O transfer, error handling and read-ahead of data. In these sub-modules, error handling module deals with the exception information thrown by the master device when an error is occurred in the process of accessing slave device. According to the characteristics of UM_BUS, exception information fall into two types, one is that the slave device is unreachable, if it happened, all buffer blocks of this device should be set to invalid state; The other type is that a device I/O operation is timeout, for example, an I/O congestion is generated at the time of accessing the device's data. In this case, nothing should do with the cache except return the error status simply. I/O transfer module is used to complete the read/write operations of devices, just as previously described, the module includes two methods to access the device's data. When the I/O request is marked with a flag that accessing the device directly and the device's data is cacheable, it is necessary to synchronize the dirty blocks in cache with this device and set the related blocks to an outdated state, and then access the device directly. Read-ahead module read several adjacent data blocks of a slave device before they are actually requested. In most cases, read-ahead significantly enhances the access performance of device, because it lets the device handle fewer commands, each of which refers to a larger chunk of adjacent blocks. Moreover, it improves system responsiveness. A process that is sequentially reading a device does not usually have to wait for the requested data because it is already available in cache. Therefore, read-ahead is conducive to improve the bandwidth utilization of UM_BUS. However, read-ahead is of no use when a process performs random accesses to devices [7-10]; in this case, it is actually detrimental because it tends to waste space in the cache with useless information. Therefore, the kernel reduces or stops read-ahead when it determines that the most recently issued I/O access is not sequential to the previous one. In this paper, we draw on the read-ahead algorithm in Linux [4] and simplify it to satisfy our requirement. The simplified read-ahead algorithm is described as below: if a read request issues as the first one read command to an appointed device, the device will output the demand block as well as several adjacent blocks (referred to as a group), such pre-fetch process is called synchronous read-ahead. If the request that in the second reading the device's data misses in the cache, which means that the access to the device is not sequential, synchronous read-ahead will be continued adopted (in other words, the number of pre-read blocks stay unchanged) and the device is marked with non-sequential flag; if the request hits,

Journal of Theoretical and Applied Information Technology
20th April 2013. Vol. 50 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

that is, the requested data block in the group of the last pre-read, it indicates that the access to the device is sequential, then marked the device with sequential flag. At the same time, the number of pre-read blocks will be double in the next operation of reading the device, such pre-fetch process is called asynchronous read-ahead. Any one cache miss happened, as well as the device is non-sequential, will return to synchronize read-ahead processing, that is, the number of pre-read block is reset to the initial value [8-10].

3.2.3 Find and replacement module

According to the device number and data offset provided by caller, the find module will searches for the cache at first and return the address of a block head to the caller if cache hit happened, otherwise, the replacement module implemented with the aforementioned LRU algorithm will be called to find a clear block or replace a block from inactive list, and then allocate this block to the caller.

3.2.4 Dirty blocks write-back module

As we have seen, this module is designed to flush dirty data in cache to the slave device. Draw lessons from the write-back mechanism of the page cache in Linux, we design this module as follow.

In this paper, the mechanisms for flushing dirty data are divided into four types for different conditions and at different times:

- Periodic write back the dirty blocks which has stayed dirty for a long time.
- If there are too many dirty blocks in the cache, a mandatory flush mechanism will be triggered to synchronize blocks with the slave device until the number of dirty blocks returns to an acceptable level.
- A process requests all pending changes of a specified device to be flushed, it does this by invoking a system call. Through this mechanism, the user can flush the data of an appointed device directly.
- If too much dirty blocks of a specified device have existed in the cache after a write operate, a balance mechanism will be called to keep the number of this device's dirty blocks under the threshold (the threshold is different for different devices). It is often trigged as a result of a massive write operation. This mechanism is conducive to save data timely and avoid mass of dirty data lost when the device becomes fault.

It should be noted that third write-back mechanism must ensure that the write-back process is complete, while others don't need to wait for the completion. The general relationship between these four mechanisms is illustrated in Figure 5 (In the figure, the writeback_dev_inodes function scan the linked list called private_list which is included in the umbus_cache structure, and then flush dirty blocks into the relevant device; The sync_dev_inode function is used for synchronizing the cache with a specified device; The do_write_blk function flushes a dirty block to the relevant device).

The first two mechanisms are implemented by means of two threads called pdflush in driver, one thread executes the periodic synchronization code, which is waked automatically by a timer defined in the umbus_cache structure. The other thread is responsible for forced flush operation and will be triggered when too many number of dirty blocks existed in the cache or failed to call the replacement algorithm (that is, there are no clear blocks in the inactive linked list).

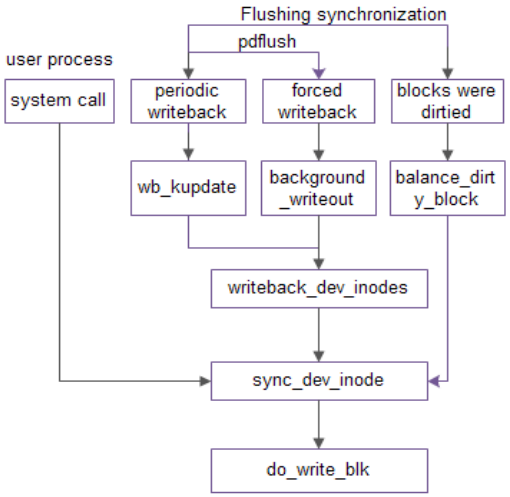

Fig5. Write-back Mechanism Figure

4. EXPERIMENTAL RESULTS

In order to verify the correctness and effectiveness of the cache, we take a UM_BUS controller as a PCI device in PC, and develop a driver, in which implemented the data cache mechanism, for this device in the windows xp platform. In our experiment, the data cache is fixed in the size of 128KB and every block is 1kB. Besides, we design a software, shown in Figure 6, to control the cache mechanism and make test more easily and accurate. After a large number of experimental tests and results analysis, we prove that the date cache can buffer the salve device's data effectively and timely write back dirty data to the device correctly, cache replacement algorithm

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

also work validly. All of these results demonstrate that the data cache mechanism is correct.

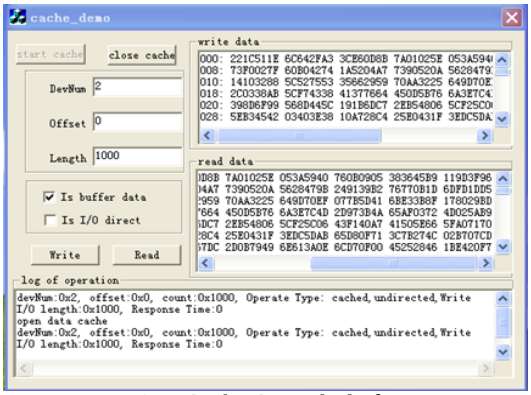

Fig6. Cache Control Platform

There are two major factors decide the performance of cache: cache speedup and cache hit rate. As the design and implementation of a date cache is the primary focus in this paper, the impact of cache hit rate will be neglected, we focus our attention on the influence of cache speedup. Therefore, it is assumed that the size of cache and hit rate is certain in our experiment. The formula of cache speedup is R=Tm/Tc. Tm is the access time of cache, in fact, it is the access time of memory; Tc is the access time of slave device. The access time of memory is certain and is 10ns in our experiment. Based on a large number of experimental tests, we get the average access time of slave device in UM_BUS is $10 \pm 2\mu s$. It is clear that the cache speedup is promoted an order of magnitude. These test results prove that the cache mechanism in this paper can accelerate the slave device's access efficiency, and therefore demonstrate the effectiveness of the cache.

5. CONCLUSIONS AND FURTHER WORK

This paper focuses on the problem of low bandwidth utilization in UM_BUS. The goal is to find an effective solution to resolve this problem. Through researching and analyzing the page cache of Linux and the feature of UM_BUS, we propose a data cache mechanism which is suitable for UM_BUS, as well as design and implement it in the driver of UM_BUS controller. It not only solves the aforementioned problem, but also makes up for the defect of cache proposed in [5] which only mergers the I/O commands and not buffers the device's data. The cache runs in UM_BUS very well. Analyzing the factors that affect the performance of the cache and cache hit rate will be our future work, and we will improve the cache according to the study results. In addition, the level of improvement to UM_BUS with the data cache still needs further research.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation. NO: 61170009.

REFRENCES:

[1] X.Zhang, X. Zuo, "Analysis of Linux Page-cache and Influence on Disk I/O Optimization", *Computer and Modernization*, Vol. 54, No. 2, 2010, pp. 106-114.

[2] Q.Yang, X. Cui, B. Zhou, "Study and Implementation of Storage Management Mechanism for Transactional File System", *Aeronautical Computing Technique*, Vol. 41, No. 5, 2011, pp. 81-84.

[3] J. Liu, X. Yang, Y. Tang, "Driver Cache: A New Cache to Improve Disk Performance", *Computer Engineering*, Vol. 30, No. 15, 2004, pp. 62-63.

[4] H.Zhu, H. Dai, Y. Yan, "Summarization on Page Replacement Algorithms for Flash Memory Storages", *Journal of Computer Research and Development*, Vol. 48, No. Suppl, 2011, pp. 251-257.

[5] H. Jin, K. Wang, "Efficient LRU algorithm for cache scheduling in a disk array system", *International Journal of Computers &amp*, Vol. 22, No. 3, 2000, pp. 134-139.

[6] G. Jia, X. Li, C. Wang, "Cache Promotion Policy Using Re-reference Interval Prediction", *Cluster Computing (CLUSTER), 2012 IEEE International Conference on* , September 24-28, 2012, 2012, pp. 534-537.

[7] F. Wu, "Sequential File Prefetching in Linux", *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. 2010, p. 218-236.

[8] Kyung-Ho Kim, Seung-Ho Lim, Kyu-Ho Park, "ADAPTIVE READ-AHEAD AND BUFFER MANAGEMENT FOR MULTIMEDIA SYSTEMS", *Eighth IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA 2004)*, 2004, pp. 264-269.

[9] Dorota M. Huizinga, Saurabh Desai, "Implementation of informed prefetching and caching in Linux", *International Conference on Information Technology: Coding and Computing (ITCC 2000)*. 2000, pp. 443-448.

[10] Linux man page for read-ahead system call, Available: http://www.kernel.org/doc/man-pages/online/ pages/man2/readahead.2.html