Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 362–371

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

GPU-Assisted Buffer Management

Jianlong Zhong Bingsheng He[1]

Nanyang Technological University, Singapore

Abstract

Cloud computing has become an emerging virtualization-based computing paradigm for various applications such as scientific computing and databases. Buffer management is an important factor for the I/O performance of the virtualized platform. In this study, we propose to leverage the memory and the computation power of the graphics processors (GPUs) to improve the effectiveness of buffer management. GPUs have recently been modeled as many-core processors for general-purpose computation. Designed as co-processors, they have an order of magnitude higher computation power than CPUs, and have a large amount of GPU memory, connected to the main memory with the PCI-e bus. In particular, we present two approaches of GPU-assisted buffer management, namely *GRAM* and *DEDU*. GRAM utilizes the GPU memory as additional buffer space and models the main memory and the GPU memory as a holistic buffer, whereas DEDU performs GPU-accelerated de-duplication to increase the effective amount of data pages that can fit into the extended buffer. We optimize both approaches according to the hardware feature of the GPU. We evaluate our algorithms on a workstation with an NVIDIA Tesla C1060 GPU using both synthetic and real world traces. Our experimental results show that the GPU-assisted buffer management reduces up to 68% of I/O cost for the traces generated from Xen.

Keywords: GPGPU, Memory Management, Virtualization, Deduplication, Cloud Computing

1. Introduction

Cloud computing (also known as utility computing) has emerged as a new computing paradigm for various applications, including scientific computing, databases and service-oriented computing. One of the key techniques for cloud computing is virtualization. Through virtualization, multiple virtual machines concurrently run on a physical machine, sharing the computation, memory and network resources. Memory management (or buffer management) has been a hot research topic for virtualization [1, 2, 3]. Previous studies have shown that the content redundancy becomes a new dimension for memory management. In the virtualized environment, there exists considerate data redundancy among the operating system images (especially for those virtual machines with the same operating system) and from the applications. In this study, we investigate how the existing hardware in the commodity machine improves the efficiency and the effectiveness of memory management, especially for the virtualized environment.

Memory management is a core and important component in the systems when the data cannot fit into the main memory. Clearly, effective buffer replacement policies [4], large buffer sizes and main memory compressions [5] are important issues in improving the buffer effectiveness. Given a fixed-sized buffer, researchers basically focus on two

[1]Email: bshe@ntu.edu.sg

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.038

aspects of buffer management: predicting the set of hot pages, and packing more pages into the buffer. Dozens of replacement policies have been proposed to capture the set of hot pages by their recency and frequency in the history, e.g., LRU (Least Recently Used) and LIRS [4]. Recently, there have been proposals on replacement polices for multi-level memory hierarchy [6] and for new devices such as solid state drives [7]. To pack more pages into the buffer, main memory compressions [5, 8] are the common methods to reduce the data size. Due to the computational overhead, main memory compressions often require special and often costly hardware [9], or with a coarse granularity. For example, in VMware ESX Server, the default memory scan for compressions is once an hour, with a maximum of six times per hour. Encouraged by the success of GPGPU (General-Purpose Computation on Graphics Processing Unit), we consider leveraging the GPU to alleviate the computational overhead and to extend the buffer size.

GPUs, originally designed for graphics rendering and gaming applications, have recently evolved into many-core processors for general-purpose computation, and GPGPU has become an active research area [10]. The GPU has over an order of magnitude higher computation capability than the CPU in terms of GFLOPS (Giga Floating Point Operations per Second). This is because the GPU devotes the majority of the die area for execution logics, whereas the major die area of the CPU is contributed to the cache. Moreover, due to the recent introduction of 64-bit addressing on the GPU, the GPU has a large piece of video memory. For instance, the NVIDIA Tesla M2070 has 6GB RAM, which can be larger than the main memory in some machines. While the data in the GPU memory is accessed via the PCI-e bus, the access latency is still over two orders of magnitude lower than the disk access (Section 2.1). When there are few rendering tasks for the GPU, the GPU is underutilized. The free GPU memory can be used as extra buffer space for storing the buffer pages, and the GPU computation power can be leveraged to perform compressions.

In order to realize the GPU-assisted buffer management, we need to manage two memory areas, connected with the PCI-e bus. Since the PCI-e bus does induce latency to the page access, hotter pages should be kept in the main memory buffer other than the GPU buffer. Moreover, the PCI-e bus is usually a significant factor in the performance of GPGPU applications [11]. In particular, we are facing the following challenging design issues:

1. What data should be stored in the GPU memory? As an extended buffer, we should consider the recency and the frequency of data pages to maximize the utility of the GPU buffer.
2. How to handle the page accesses in the GPU buffer, such that the bandwidth of the PCI-e bus is fully utilized? A data transfer on the PCI-e bus involves a fixed cost of initializing the bus. The page accesses to the GPU buffer should be designed with this overhead in mind.
3. How to efficiently compress the data pages at runtime? Compressions are not without drawbacks. While compressions can reduce the memory space required for the same amount of data, they introduce the runtime computational overhead. Moreover, some data types, such as those have been compressed, cannot be effectively reduced by compressions. Thus, we must develop a compression algorithm to balance between the gain of reduction on the data size and the runtime overhead.

Attempting to address these issues, we have developed two GPU-assisted buffer management algorithms, called *GRAM* and *DEDU*. GRAM manages all the pages within the GPU buffer and the main memory buffer in a uniform buffer management policy. For example, LRU manages all the pages in a single LRU queue, consisting of pages from the main memory buffer and the GPU buffer. Due to the data transfer cost of the PCI-e bus, the GPU buffer is designed to be an eviction cache. Moreover, the page accesses to the GPU buffer are performed in a batched way, so as to amortize the cost of the PCI-e overhead. The batch size is a tuning parameter for GRAM to balance the response time and the throughput. Since all the pages in the extended buffer are managed by the same replacement policy, the fruitful results in the buffer management research are relevant to GRAM.

Enhancing GRAM with data compressions, DEDU further packs more pages into the extended buffer with deduplications [12]. Deduplication has demonstrated to be useful in reducing the effective data size for archives [13] and memory management of virtual machines [1]. In our implementation, a page is divided into multiple blocks of the same size. Deduplication is applied to remove the redundancy among data blocks. Since the blocks are with the same size, deduplication well fits the GPU computation model. Moreover, deduplication is highly parallelizable, with intra- and inter-block parallelism. Due to the massive computation power, the GPU supports a high throughput of deduplications, and the most time consuming part of the deduplication is off-loaded from the CPU to the GPU.

We conduct experiments on a NVIDIA GPU of 240 cores and 4GB device memory. We evaluate the effectiveness of GRAM and DEDU with synthetic workloads in order to fully control the parameters to assess the design of GPU-assisted buffer management. The results show that: a) batched processing significantly improves the bandwidth

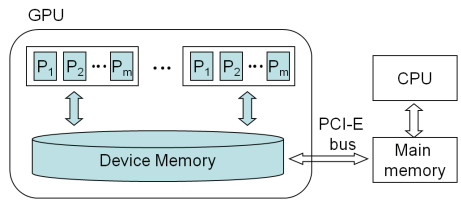


Figure 1: Architecture of GPU.

utilization of the PCI-e bus, with eight times improvement; b) GRAM achieves up to 83% smaller I/O cost per page access than the native main memory buffer for our synthetic workloads; c) with deduplication, DEDU can further reduces 81% I/O cost per page access than GRAM. Moreover, our case studies with traces from Xen demonstrate a reduction of 68% on the average I/O cost.

Organization. The remainder of this paper is organized as follows. Section 2 reviews the related work on GPGPU and buffer management. We present the detailed algorithm for GRAM and DEDU in Sections 3 and 4, respectively. We present the evaluation results in Section 5, and conclude in Section 6.

2. Preliminary and Related Work

2.1. GPGPU

Graphics processors (GPUs) can be modeled as many-core processors. The architecture of the GPU is illustrated in Figure 1. The GPU is programmed as a co-processor to the CPU, via a PCI-e bus. The GPU consists of many multi-processors. A multi-processor has multiple cores, executing instructions in the SIMD (Single Instruction Multiple Data) fashion. That is, all the cores within the same multi-processor execute the same instruction on different inputs simultaneously. Different multi-processors execute the same program in the SPMD (Single Program Multiple Data) fashion. The program is called *kernel* (in NVIDIA CUDA's term [14]). Multiprocessors share the device memory, which has a high bandwidth and a high access latency. For example, NVIDIA Tesla C1060 GPU has a device memory of 4 GB, a bandwidth of 73.2 GB/s, and a latency of around 400 cycles. The GPU memory has been increasing in recent years. For example, a new generation NVIDIA Tesla M2070 GPU has 6 GB RAM.

As a coprocessor, the GPU executes a kernel in three basic steps. First, the input data to the kernel are copied to the GPU device memory from the main memory via the PCI-e bus. Second, the GPU executes the kernel in parallel on the multi-processors. Third, the result is copied from the device memory to the main memory.

Although the PCI-e bus usually has a lower bandwidth than the main memory, the data transfer on the PCI-e bus is much faster than the disk accesses. Each data transfer involves a fixed cost of an initialization of PCI-e transfer and the data transfer cost. The initialization cost is small (e.g., around 0.015 ms in our experiment). The bandwidth of PCI-e has been improved recently. The PCI-e2 has a theoretical bandwidth of 8 GB/sec. Copying a 4KB page costs 0.0155 ms, which is over two orders of magnitude faster than fetching a page from the disk.

Due to the superb computation power and memory bandwidth, the GPU has been used as a powerful accelerator for various applications, including scientific computing [15, 16, 17] and databases [11, 18, 19, 20]. There have been projects such as Folding@Home utilizing free GPU resources. Recently, Amazon has offered GPU-enabled EC2 virtual machines for high performance computing. We utilize the GPU-optimized data parallel primitives such as map and prefix scan [21, 22] as building blocks for the GPU-based deduplication. In contrast with the previous work on a GPU-accelerated file archives [13], this study focuses on off-loading the most time consuming part of deduplication to the GPU for buffer management at runtime. We refer readers to a survey on the GPGPU [10] for more details.

2.2. Buffer Management

Buffer management is an active research area for many systems such as virtualization, databases and operating systems. Most algorithms have focused on minimizing the buffer miss rate. The theoretically miss-rate-optimal replacement policy, known as Belady's algorithm [23], is to evict the page whose next use will occur farthest in the future. The policy most widely used by commercial systems is LRU and its variants [24, 4]. LRU always evicts the least recently used page.

Due to the increasing speed gap between the processor and the disk, memory hierarchy has been become deeper than before. New memory levels and new buffer management algorithms have been invented [7, 6]. Applying the single-level algorithm to each level results in sub-optimal performance. Multi-level caching algorithms become popular, since the locality of the page accesses at the lower level is much weaker than those in the higher level. Due to the weaker locality, new replacement policies have been proposed, including coordinating the replacement polices among multiple levels [25], giving hints to lower level buffer management [26], and specific second-level buffer management algorithms [6]. This study proposes to extend the main memory buffer with the GPU memory. Since the GPU is not directly connected with the disk, it is not designed as an intermediate level between the main memory and the disk. Instead, it is designed as an eviction cache, managed with the same replacement policy as the main memory buffer.

Main memory compressions has been studied for buffer management [5, 8]. These methods use a software cache to store data in compressed format. A hardware compression/decompression unit has been proposed to perform the compressions between cache and RAM [9]. Data is stored uncompressed in cache, and compressed on-the-fly when transferred to memory. Compression is usually done off-line and can be slow, while decompression is done during execution, with the special hardware. Overall, the software-based approaches usually pose significant computational overhead to the CPU, and the hardware-based approaches need costly hardware. In contrast, this study uses the commodity hardware, GPU, as a co-processor to perform the data compression. Along the line of memory compressions, page sharing among different virtual machines has been actively studied [1, 2, 3]. These studies dynamically identify the redundancy among data pages in the main memory, and remove the redundancy for an improved memory usage. Through offloading the deduplication computation to the GPU, our GPU-based solution can be applied to those existing virtualization techniques.

3. GRAM: Extending Buffer with GPU Memory

The basic scheme, GRAM, is to use the GPU memory as an extended buffer area for holding more pages, thus reducing the number of costly disk page accesses.

Compared with main memory buffer, the GPU buffer is an extended buffer, where pages are accessed indirectly. The page needs to be transferred via the PCI-e bus. Thus, a page access in the main memory buffer is faster than that in the GPU buffer. GRAM manages the memory space from both the GPU buffer and the main memory buffer with a uniform buffer management policy. Since LRU and its variants are widely used in the buffer management, we use LRU in this study. The GPU buffer and the main memory buffer have their own LRU queues. These two queues virtually compose an LRU queue, where the head part is from the LRU queue in the main memory buffer, and the tail part is from the LRU queue in the GPU buffer.

With the GPU buffer as an eviction cache, GRAM handles a page request as follows. When a page request comes to the extended buffer, the main memory buffer checks whether the page access is a hit or a miss. If it is a hit, the LRU queue in the main memory buffer adapts to this hit and the page is returned to the application. Otherwise, the page access is redirected to the GPU buffer. If the page is in the GPU buffer, the page is copied from the GPU memory to the main memory via the PCI-e bus. Since the GPU buffer is designed as an eviction cache to the main memory buffer, the page is removed from the GPU buffer. If the main memory buffer is already full, we need to perform page replacement, and the victim page is inserted into the GPU buffer via the PCI-e bus. The victim is inserted into the head of the LRU queue in the GPU buffer. Thus, the LRU queues in the main memory buffer and the GPU buffer form a virtual LRU queue.

Finally, if the page cannot be found in the GPU buffer, a disk page I/O occurs. When the disk I/O completes, the page is loaded into the main memory buffer. If the buffer replacement occurs in the main memory buffer, the victim will be added to the GPU buffer, which may further trigger buffer replacement on the GPU buffer. The buffer replacement on the GPU buffer needs special care if the evicted page is dirty. The dirty page needs to be transferred back to the main memory, and then is written back to the disk.

An example of handling page accesses is illustrated in Figure 2. Two LRU queues are maintained in the extended buffer manager. Both the GPU buffer and the main memory buffer have the capacity of three pages. For the first access on page A, it is a miss for the main memory buffer, but a hit on the GPU buffer. Thus, pages A and F are exchanged. For the access on page G, neither of the buffer pools has the page. A page is fetched from the disk. Page E is evicted from the main memory buffer, and is inserted into the GPU buffer. Next, Page C is evicted from the GPU buffer.

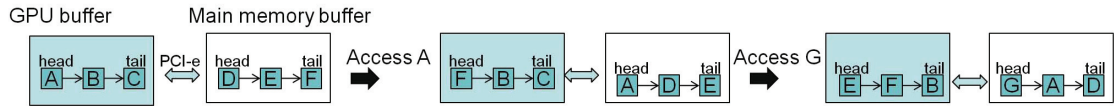


Figure 2: GRAM: extending the main memory buffer with the GPU memory.

To implement these two LRU queues as a virtual LRU queue, a hash table is maintained in the main memory, mapping the physical page ID to the logical page ID for all pages in the extended buffer. We use the most significant bit in the logical page ID to denote whether the page belongs to the GPU buffer or the main memory buffer. A bit of one means that the page belongs to the main memory buffer, and the GPU buffer otherwise.

Batched Page Miss Handling. Handling the page requests individually on the GPU buffer can be costly, since each data transfer induces a fixed cost of initialization cost. The PCI-e bus is severely under-utilized. We propose to perform batched page miss handling for amortizing the initialization cost. To improve the response time of batched page miss handling, we adopt a time-out mechanism to flush the page misses at the period defined by users. Page requests on the GPU buffer are performed in a batch when the number of page misses in the main memory buffer exceeds a predefined threshold, or when the predefined epoch expires.

The batched page miss handling algorithm works in the following four steps. First, given a batch of page requests (in the form of an array in the implementation), we gather all the pages to a temporary area B on the GPU. The gathering can be done with the device-to-device data copy API or a kernel execution. In the former case, each page requires an API call, and the later requires only a single kernel call. All the data copies are performed within the GPU for high speed. Second, the page contents in the temporary area B are copied to a temporary area B' in the main memory. This is the data transfer via the PCI-e bus. Multiple pages are transferred with a single transfer, amortizing the overhead of initializing the PCI-e bus. Third, the pages in B' are added to the main memory pool, and evicted pages are stored in a temporary area B''. Fourth, the pages in B'' are transferred to the GPU memory via a single PCI-e call, and are added to the GPU buffer. The batched page miss handling improves the utilization of the PCI-e bandwidth. For the dirty victims from the GPU buffer, we need to write the pages to the disk, via PCI-e data transfer to the main memory.

4. DEDU: Buffer Management with GPU-Assisted Deduplication

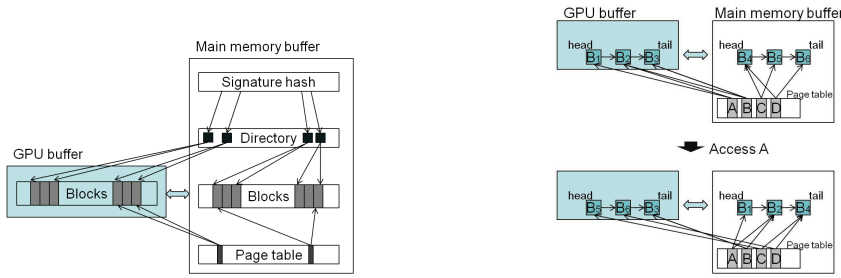
As a coprocessor, the GPU not only has a large piece of device memory, but also has massive computation power. GRAM exploits the device memory only to extend the buffer management. With the computation power of the GPU, we can further increase the effectiveness of the buffer management with memory compressions. In this section, we present our deduplication based buffer management named DEDU.

Data redundancy is common in the cloud, for example, an email can be stored multiple times in the system and virtual machines run with the same operating system. In data archive, a common technique to reduce data redundancy is data deduplication, which identifies the duplicate blocks/files and stores only one copy of them. Deduplication is usually based on hashing, avoiding costly content comparison for every pair of blocks. However, hashing is still costly for per page access. In DEDU, the GPU is used for the hashing computation, posing little computation overhead to the CPU-based execution.

4.1. Buffer Management Policy

Before diving into the details of buffer management policy, we first study how the deduplication can be integrated into the buffer management. A page is divided into D disjoint regions of the same size, K KB. We call D as the degree of deduplication. A region is called a *block*. The degree is a tuning parameter to balance the probability of finding duplicate blocks, and the efficiency of deduplication. If the degree is high, the block size is small and more blocks can be found to be duplicates, but with more computational overhead.

Each block has a signature, i.e., a hash value calculated with advanced hash functions with few collision conflicts. We use the encryption algorithm MD5 (Message-Digest algorithm 5). The hash value is used for speedup the process of deduplication. If two blocks have different signature values, they are different. Otherwise, we need to examine their contents in order to make sure whether they are duplicates or not.



(a) DEDU (b) Example ($D = 2$)

Figure 3: DEDU: extended buffer management with deduplication.

The deduplication is performed at runtime. Each block needs calculating its signature only once. If the page is updated, the signature is recalculated. The blocks and their signatures are maintained in a hash table. We implement it as a bucket chained hash table, as illustrated on the right part of Figure 3 (a). There are three indirection layers in the hash table:

- Signature hash. The signature hash is to map all signatures to their positions in the directory (the second level).
- Directory. The directory is to map the signature to the blocks with the same signatures. Each directory entry is represented as the start and end positions in the third level.
- The block array. All the blocks are packed into an array. The blocks with the same signature values are stored consecutively in the array, corresponding to the start and the end positions for each directory entries.

This three-level directory structure speeds up the searching process of deduplication. The block array is the major memory consumer of the extended buffer. The signature hash and the directory are relatively small, compared with the block array. In our implementation, the total size of signature hash and directory is smaller than 1% of the block array.

Arrays are the basic building components for the data structure on the GPU. We consider packed memory array (PMA) as a storage model for dynamic data [27]. The basic idea is to leave some vacant slots in the array. When a new block needs to be inserted into the block array, we try to find a vacant slot within the region specified by the start and end positions. If there is no vacant slot, we need to re-organize its neighboring regions in order to find one. This is to amortize the update cost, by reducing the number of total re-organizations on the entire array. Note, we also perform batched page miss handling for an improved performance.

Figure 3 (a) shows the data structure in DEDU. The signature hash and the directory are stored in the main memory. Blocks are stored into two arrays, one in the GPU buffer, and the other in the main memory buffer. A page table is maintained, mapping the logical page ID to its D block pointers. We use the highest bit of the block pointer as a flag indicating whether it is in the main memory buffer or in the GPU buffer. Note, a page can consist of blocks in the GPU buffer and those in the main memory buffer.

The page access is via looking up the page ID in the page table. If it is found, the D blocks assemble the page. With deduplication, the granularity of data accesses in DEDU is a block. A page read is to read all the D blocks, following the pointers in the page table. If all the blocks are in the main memory, the page access is a hit in the main memory buffer. Otherwise, it is a miss, and the missing block requests are redirected to the GPU buffer.

The blocks in the buffer pool are managed in an LRU queue. Thus, both the GPU buffer and the main memory buffer manage the blocks according to LRU. The process is similar to the GRAM process. When a page hit occurs on the main memory buffer, the LRU queue of blocks are adjusted and the corresponding D blocks are moved to the head. Otherwise, if it is a hit on the GPU buffer, we need to copy the blocks between the GPU buffer and the main memory buffer. Note, the batched page miss handling is adopted. The major difference is the cases where a page is updated, or where a disk I/O occurs. In both cases, new blocks are added to the buffer, and we need to maintain the deduplication structure. First, the signature of the new block is calculated on the GPU. Second, it is inserted into the hash structure. An example is illustrated in Figure 3 (b).

We reduce the number of deduplication operations in two ways. First, we maintain a page signature map in the main memory to record the signatures for the pages that have been accessed. If the workload is read-only, each page requires the hashing calculation at most once. Second, if the number of deduplication operations after applying the first optimization exceeds the computation capability of the GPU, we directly maintain the blocks belonging to the page without deduplication. For those blocks, DEDU degrades to GRAM.

4.2. GPU-based Deduplication

MD5 is a widely used cryptographic hash function. It calculates a 128-bit hash signature for the input data. The input to MD5 requires the chunks of 512-bit blocks. We ensure each block is multiple times of 512 bits, thus performing MD5 without padding.

The basic version of MD5 has the following three steps.

1. Initialize the 128-bit signature in four 32-bit integers h_i ($0 \leq i \leq 3$).
2. For each 512-bit chunk in the input block, the algorithm performs a series of mathematic transformations and bit operations. There are 64 iterations of such computations, and thus it is the most computation intensive part of MD5. We refer readers for more details of MD5 to RFC 1321 [28]. The output of this stage of each chunk is four 32-bit integers a_i ($0 \leq i \leq 3$), and then adds a_i to h_i.
3. Output the signature of the input block as the concatenation of h_i ($0 \leq i \leq 3$).

The basic version of MD5 well fits for the GPU computation pattern. For the second step, chunks within the same block are handled independently, and different blocks are also executed in parallel. The algorithm is executed on the GPU with full parallelism. The third step is calculated with a prefix-sum algorithm [21]. All these algorithms and primitives are optimized for the GPU. In CUDA, the thread configuration is a hierarchy, allowing us to easily configure the threads for both intra- and inter-block parallelism.

4.3. GRAM vs. DEDU

We have developed two GPU-assisted buffer management algorithms, GRAM and DEDU. They have their own strength and weakness. GRAM is simple, thus with little runtime overhead. DEDU exploits runtime deduplication techniques to increase the effective size of data cached in the buffer. Thus, DEDU have a relatively higher runtime overhead than GRAM. There are several issues affecting the comparison between GRAM and DEDU. One of the important factors is the *deduplication factor*, where the deduplication factor is defined to be the ratio of the raw data size over the data size after deduplications. If the deduplication factor is close to one, deduplication has little impact on the input data, and the runtime overhead of DEDU offsets the performance gain. Example data types are compressed data or data archives after deduplication. Another case is update-intensive workloads. When updates are intensive, the overhead of maintaining the hash structure and re-calculating the signatures can offset the performance gain of deduplication. In this case, we should choose GRAM. The choice of GRAM or DEDU is a configurable setting in the GPU-assisted buffer management.

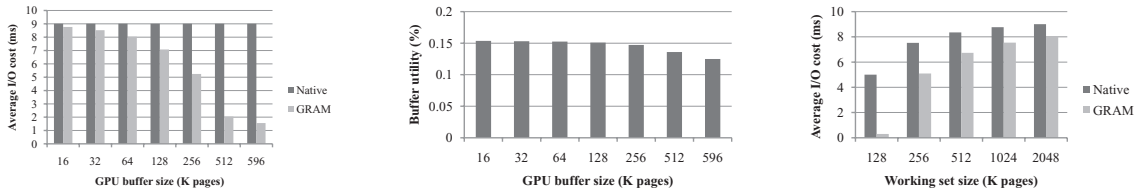
High volumes of page requests may saturate the capability of DEDU. First, the number of pages required to be transferred exceeds the limitation of the PCI-e bus. Second, the number of deduplication operations exceeds the computation power of the GPU. Deduplication occurs when there are new pages added to the buffer, or when the pages in the buffer is updated. In this case, we choose GRAM for some of the blocks without deduplication.

5. Evaluations

5.1. Experimental Setup

We have implemented and tested our algorithms on a PC with an NVIDIA Tesla C1060 GPU and an Intel Xeon Quad-Core CPU. The hard disk is a 7200rpm SATA magnetic hard disk. The GPU is with 240 cores running at 1,300MHz. Based on our measurements, the GPU achieves a memory bandwidth of around 73.2 GB/s, the CPU with 7.6 GB/s and the PCI-e bus with 3.2 GB/s.

We implement the buffer manager with NVIDIA CUDA 3.2. We have evaluated the GPU-assisted buffer management with traces generated from a popular virtualization platform, Xen, as well as synthetic traces.



(a) (b) (c)

Figure 4: Evaluations of GRAM: (a) I/O cost per page access the GPU buffer size varied; (b) buffer utility with the GPU buffer size varied; (c) with the working set size varied

We use synthetic workloads for the full control of the workload characteristics such as distributions and the read/write ratio. In particular, we have generated synthetic traces with page accesses of reads and writes conforming to uniform and skew distributions. Due to the space limitation, we mainly present the results on the uniform traces with different read/write ratios and working set sizes. The default setting is: the page size is 4K bytes; the main memory buffer and the GPU buffer contains 64K pages each, the working set is five times as large as the total buffer size (i.e., 640K pages), and the traces are read-only. The traces are executed on a large file. We use a large file to simulate the concatenation of many files. We consider the deduplication factor between 1 to 10. Given a duplication factor d, the working set of w blocks and the block size b, we generates a file of $(w \times b \times d)$, and each distinct block has d copies in the file on average. We use the deduplication factor of four by default. The buffer size in RAM is 256MB by default, and the workload footprint varies up to 8GB.

Since I/O usually is the main metric for the system performance, we use the average I/O cost per page access (ms) to compare the effectiveness of different buffer management schemes. We define the buffer utility to be the average buffer miss rate reduction per MB of GPU memory for the GPU buffer.

5.2. Results on GRAM

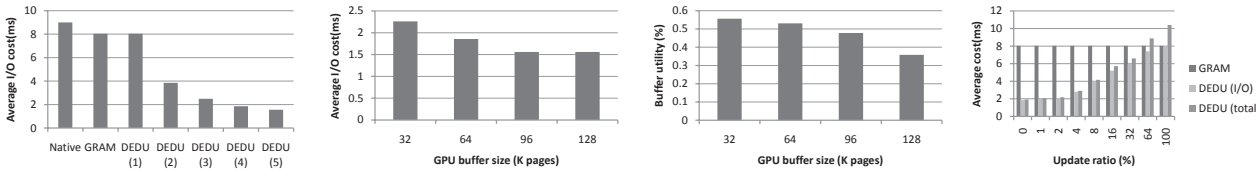
Figure 4(a) shows average cost per page access for GRAM and the native LRU (without GPU extended buffer) as the GPU buffer size varies. We compare GRAM with the Native method without GPU-assisted buffer management. The main memory buffer size is fixed to be 64K pages, and the working set size is fixed to be 640K pages. We measure the average cost for four million page accesses, randomly accessing pages in the working set. As the GPU buffer size increases, the average I/O cost per page access decreases. When the GPU buffer size is 596K pages, the entire working set fit into the extended buffer. Note, the measurement is for four million of page accesses. Further increasing the GPU buffer size does not reduce the buffer miss rate. Figure 4(b) shows the buffer utility of the GPU buffer. As the GPU buffer size increases, the buffer utility decreases. Thus, the benefit of the GPU buffer marginally decreases. In all these settings, the overhead of the PCI-e transfer overhead is less than 1% of the I/O cost, since the I/O cost is dominant. The PCI-e cost could become significant in the long-term execution when the I/O cost is not significant (e.g., the working set fits into the extended buffer).

By reducing the number of PCI-e bus data transfer operations, the batched handling method achieves a significantly lower cost per page transfer, with a reduction of over 87%. By reducing the number of calling device-to-device memory copy APIs, the kernel-based method achieves up to 67% of reduction in the cost of the API-based method. When the number of pages in a batch exceeds 256, the cost per page transfer for the kernel method becomes stable. We use this batch size as the default setting in our experiment to balance the bandwidth utilization of the PCI-e bus and the response time of the requests in a batch. The peak bandwidth of the batched page miss handling is 2.1 GB/sec, which achieves around 66% of the peak bandwidth of the PCI-e bus.

Figure 4(c) shows the average I/O cost as the working set size increases. The GPU buffer and the main memory buffer sizes are both fixed to be 64K pages. When the working set size is 128K pages, the entire working set fits into the extended buffer, the performance improvement is maximized. As the working set size increases, the performance gain of GRAM decreases. The performance improvement is up to 90%.

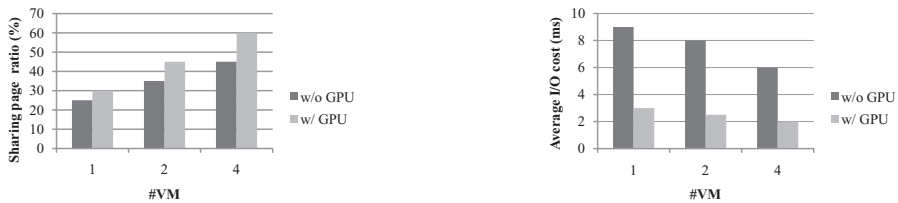
5.3. Results on DEDU

Figure 5(a) shows the average I/O cost per page access with different deduplication factors. "DEDU(f)" denotes the measurement for DEDU when the input data with a deduplication factor of f. The working set size is five times



(a) (b) (c) (d)

Figure 5: Evaluations of DEDU: (a) I/O cost per page access with the deduplication factor varied; (b) I/O cost per page access with the GPU buffer size varied; (c) buffer utility of DEDU with the GPU buffer size varied; (d) I/O cost per page access with the update ratio varied.



(a) Sharing page ratio (b) Average I/O cost

Figure 6: Impact of GPU-assisted buffer management varying the number of virtual machines.

as large as the total buffer size of the main memory buffer and the GPU buffer. When the duplication factors is equal to or larger than five, the data fit into the buffer. We can see that the average I/O cost decreases, as the deduplication factor increases. When the deduplication factor is one, DEDU has the similar average I/O cost to the GRAM (i.e., 5% higher than GRAM). Note, the GPU-based hashing significantly improves the performance of deduplication. The speedup of the GPU only algorithm over its CPU-based counterpart is 24–29 (the results on evaluating the GPU- and the CPU-based deduplication are omitted).

We further examine the buffer utility of different GPU buffer sizes with DEDU. Figure 5(b) shows the average I/O cost with the GPU buffer size varied. The duplication factor is fixed to be 4. As expected, as the GPU buffer size increases, the average I/O cost decreases till the working set fits into the extended buffer. Figure 5(c) shows the buffer utility of the GPU buffer. We observe a marginally decreasing trend as the GPU buffer size increases. Compared with GRAM, DEDU has a higher buffer utility, i.e., the same amount of the GPU memory gaining a larger reduction on the buffer miss rate. Clearly, deduplication helps packing more data pages into the same amount of memory, resulting in a higher utility.

We investigate the performance impact of updates in the workload. Figure 5(d) shows the average I/O cost per page access of DEDU and GRAM as the update ratio varied. The cost of GRAM is stable, whereas the I/O cost of DEDU increases as the update ratio increases, due to the reduction in the duplication factor. We also show the average cost per page access, including the I/O cost and the computation cost. As the update ratio increases, the average cost per page increases dramatically, due to the overhead of more deduplication operations. We observe a cross point between GRAM and DEDU (i.e., the update ratio is between 32% and 64%). In practice, if the workload consists of updates more than the cross point, we use GRAM; otherwise, we use DEDU. We are investigating a fine-grained adaptation between GRAM and DEDU by categorizing the pages into read- and write-intensive ones.

5.4. Case Studies

We evaluate the buffer manager with real traces collected from running Xen. We varied the number of virtual machines running on the physical machine. Each virtual machine runs httperf benchmark [29] on Fedora 9 (Linux 2.6.25-14), and is provisioned with RAM 1GB. We collected the traces from Xen, and replayed the trace on our buffer manager. Figure 6 shows the *sharing page ratio* and the average I/O cost with and without GPU assistance. The sharing page ratio is defined to be the ratio of pages with sharing. With the GPU memory, the sharing page ratio increases up to 40%. With the combined effect of GPU computation and the GPU memory, the average I/O cost reduces up to 68%.

6. Conclusions

The emergence of cloud computing raises the recent research interests on memory management for virtualization and other systems. In this paper, we leverage the available memory and computation resources of the GPU to assist the buffer management. The extended buffer space and the deduplication-based online compression improve the effectiveness of the buffer management. Our experimental results show that the GPU-assisted buffer management reduces up to 68% of I/O cost for the traces generated from Xen. While our work targets at virtualized platforms, we conjecture that it is applicable to general I/O systems. Our future work includes integrating the GPU-assisted buffer manager into virtualization and scientific systems.

Acknowledgements

The authors thank the anonymous reviewers for their insightful suggestions. This work was supported by an AcRF Tier 1 grant from Singapore.

References

[1] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, A. Vahdat, Difference engine: Harnessing memory redundancy in virtual machines, in: USENIX OSDI, 2008.
[2] J. Ren, Q. Yang, A new buffer cache design exploiting both temporal and content localities, in: ICDCS, 2010.
[3] G. Miłós, D. G. Murray, S. Hand, M. A. Fetterman, Satori: enlightened page sharing, in: USENIX Annual technical conference, 2009.
[4] S. Jiang, X. Zhang, LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance, SIGMETRICS Perform. Eval. Rev. 30 (1).
[5] F. Douglis, The compression cache: Using on-line compression to extend physical memory, in: USENIX Winter Conference, 2001.
[6] Y. Zhou, J. Philbin, K. Li, The multi-queue replacement algorithm for second level buffer caches, in: USENIX Annual Technical Conference, 2001.
[7] M. Canim, G. Mihaila, B. Bhattacharjee, K. Ross, C. Lang, SSD bufferpool extensions for database systems, in: VLDB, 2010.
[8] P. R. Wilson, S. F. Kaplan, Y. Smaragdakis, The case for compressed caching in virtual memory systems, in: USENIX Annual Technical Conference, 1999.
[9] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, P. M. Bland, Ibm memory expansion technology (mxt), IBM J. Res. Dev. 45 (2).
[10] J. D. Owens, D. Luebke, N. K. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, in: Eurographics 2005, State of the Art Reports, 2005.
[11] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, P. V. Sander, Relational query coprocessing on graphics processors, ACM Trans. Database Syst. 34 (4) (2009) 1–39.
[12] Deduplication, http://en.wikipedia.org/wiki/Data_deduplication (2010).
[13] X. Li, D. Lilja, A highly parallel gpu-based hash accelerator for a data deduplication system, in: Parallel and Distributed Computing and Systems, 2009.
[14] NVIDIA CUDA, http://developer.nvidia.com/object/cuda.html.
[15] Folding@home, http://folding.stanford.edu/.
[16] M. Taufer, P. Saponaro, O. Padron, S. Patel, Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs, in: IPDPS, 2010.
[17] L. Xu, M. Taufer, S. Collins, D. Vlacho, Parallelization of tau-leap coarse-grained monte carlo simulations on GPUs, in: IPDPS, 2010.
[18] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, Mars: a mapreduce framework on graphics processors, in: PACT, 2008.
[19] W. Fang, B. He, Q. Luo, N. K. Govindaraju, Mars: Accelerating mapreduce with graphics processors, IEEE Transactions on Parallel and Distributed Systems.
[20] B. He, J. X. Yu, High-throughput transaction executions on graphics processors, in: PVLDB, 2011.
[21] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for gpu computing, ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2007.
[22] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, Relational joins on graphics processors, in: SIGMOD, 2008.
[23] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems.
[24] T. Johnson, D. Shasha, 2Q: A low overhead high performance buffer management replacement algorithm, in: VLDB, 1994.
[25] S. Jiang, K. Davis, X. Zhang, Coordinated multilevel buffer cache management with consistent access locality quantification, IEEE Transactions on Computers.
[26] S. Jiang, F. Petrini, X. Ding, X. Zhang, A locality-aware cooperative cache management protocol to improve network file system performance, in: ICDCS, 2006.
[27] B. He, Q. Luo, Cache-oblivious query processing, in: CIDR, 2007.
[28] MD5, http://tools.ietf.org/html/rfc1321 (1992).
[29] Httperf, http://www.hpl.hp.com/research/linux/httperf/ (2010).