DMA Cache: Using On-Chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance

Dan Tang[1,2,3], Yungang Bao[1], Weiwu Hu[1,2], Mingyu Chen[1]
[1]Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]Loongson Technologies Corporation Limited
[3]Graduate School of Chinese Academy of Sciences
{tangdan, baoyg, hww, cmy}@ict.ac.cn

Abstract

As technology advances both in increasing bandwidth and in reducing latency for I/O buses and devices, moving I/O data in/out memory has become critical. In this paper, we have observed the different characteristics of I/O and CPU memory reference behavior, and found the potential benefits of separating I/O data from CPU data. We propose a DMA cache technique to store I/O data in dedicated on-chip storage and present two DMA cache designs. The first design, Decoupled DMA Cache (DDC), adopts additional on-chip storage as the DMA cache to buffer I/O data. The second design, Partition-Based DMA Cache (PBDC), does not require additional on-chip storage, but can dynamically use some ways of the processor's last level cache (LLC) as the DMA cache.

We have implemented and evaluated the two DMA cache designs by using an FPGA-based emulation platform and the memory reference traces of real-world applications. Experimental results show that, compared with the existing snooping-cache scheme, DDC can reduce memory access latency (in bus cycles) by 34.8% on average (up to 58.4%), while PBDC can achieve about 80% of DDC's performance improvements despite no additional on-chip storage.

1. Introduction

DMA technology provides special channels for CPU and I/O devices to exchange I/O data, and the memory is used for buffering the I/O data. When the CPU wants to handle I/O data, it triggers the DMA write operations that transfer the I/O data from I/O devices to the memory. On the opposite direction of the CPU writing data to I/O devices, the DMA read operations, i.e., transferring I/O data from the memory to I/O devices, are performed.

In the past decade, the performance of I/O bus and device has improved dramatically. New I/O bus technologies, such as PCI-Express 2.0, AMD's HyperTransport 3.0 and Intel's QPI, can provide bus bandwidths of 10~20GB/s that are close to the bandwidth of a DDR2/3 DRAM memory system. I/O device's bandwidth and latency have also substantially improved. For example, using a single server, Fusion-io Company recently adopted PCI Express and SSD technology to exceed one million IOPS (I/O Operations Per Second) and 8GB/s sustained throughput [1]. Unlike traditional hard disks, SSD avoids mechanical delays so that it usually has low access latency, e.g., experiments on Intel® X25-E SSD demonstrate that the latency of reading 4KB data is only 75us [2]. Additionally, the latency of Infiniband's RDMA operations can even be less than 1us [5]. With these substantial technology advances, moving I/O data in/out memory has become critical for DMA scheme, because it exhausts a substantial portion of memory bandwidth as well as accounts for a considerable part of I/O operation latency. Kumar et al. have profiled the execution of handling 4KB network I/O data and found that buffer copy (i.e., I/O data movement) is the largest single contributor to the overall processing time per packet, accounting for over 25% [18].

To address this problem, researchers have proposed several schemes to reduce the memory access overhead of the DMA operations. Some studies proposed the cache injection techniques that directly inject I/O data into a processor's cache [21-23]. Iyer proposed a chipset cache to improve performance of Web Servers [15]. Intel proposed a Direct Cache Access (DCA) scheme [14] that has been implemented in the Intel 82599 10 Gigabit Ethernet (GbE) controller [4]. Although these schemes have demonstrated significant I/O performance improvement for a few specific workloads under some specific environments, their evaluation results are limited because of the three reasons: (1) Little previous work investigated the inherent characteristics of DMA memory references, such as DMA request size and reuse distance (the definition is in Section 2). The characteristic differences between DMA and CPU memory reference patterns, however, are significant for studying effective optimization schemes. (2) Their studies mainly focus on specific processors that are designed for some specific applications, such as networking processors and embedded processors. (3) Their conclusions are usually drawn from studying one type of I/O such as network. In real systems, e.g., Video-on-Demand servers and data centers, however, various types of I/O operations can

Corresponding author: Weiwu Hu (hww@ict.ac.cn).

exist simultaneously. Moreover, different applications can also combine different types of I/O. Thus the performance of the existing optimization approaches may be sensitive to different applications. Leon et al. have found that cache injection approaches can provide significant performance benefits, but it can also hurt some applications' performance [19].

In this paper, by analyzing the memory traces of various types of I/O operations, e.g., reading/writing disk data and transmitting/receiving network packets, we obverse the different characteristics of DMA and CPU memory reference behaviors. For example, 1) the I/O data's reuse distances in the "DMA-Produce-CPU-Consume (DPCC)" direction are much smaller than the CPU data's reuse distances, but the I/O data's reuse distances in the "CPU-Produce-DMA-Consume (CPDC)" direction are very large; 2) the DMA memory references are very regular, usually being linear within a DMA buffer. In spite of the substantially different characteristics, most previously proposed approaches still unify I/O data and CPU data in a shared cache. Inspired by the idea of separating instruction and data, we believe that separating I/O data from CPU data is more likely to leverage the inherent characteristics of DMA memory reference patterns for improving I/O performance than unifying them in one cache.

We propose a DMA cache technique, which uses dedicated on-chip storage to store I/O data, to architecturally separate I/O data from CPU data. Nevertheless, the dedicated DMA cache imposes many design challenges, especially the cache coherence issue. To address the I/O data coherence challenge, we have refined both the MOESI [3] and ESI [16] cache coherence protocols for the DMA cache. This paper presents two concrete DMA cache designs for various purposes. The first design, Decoupled DMA Cache (DDC), adopts dedicated on-chip storage as the DMA cache and is suitable for IO-specific processors. DDC has little design complexity, because it avoids modifying processor's last level cache (LLC) controller. The second design, Partition-Based DMA Cache (PBDC), can dynamically use some ways of a processor's LLC as the DMA cache so that it is suitable for general purpose processor. Owing to no additional on-chip storage, the PBDC's design cost is much lower.

By using an FPGA-based emulation platform and the memory reference traces of real-world applications including file-copy, SPECWeb2005 and TPC-H, we have implemented and evaluated the two DMA cache designs as well as several previous unified approaches. The experimental results show that although the previously proposed share-cache scheme has high cache hit rates for I/O data, its policy of unifying I/O and CPU data might decrease performance (by up to 15.1%, the baseline being the memory access latency of the traditional snooping-cache scheme) because of cache interference. In contrast, while separating I/O data from CPU data, DDC can reduce the memory access latency (in bus cycles) by 34.8% on average (up to 58.4%), which is about 2.5X of the Prefetch-Hint scheme [18]. Despite no additional on-chip storage, PBDC can achieve about 80% of DDC's improvements. Additionally, we find that when using the write-through policy for I/O data, I/O performance can be further improved for all approaches we have evaluated. The reason is that the write-back policy can convert the contiguous DMA memory references into non-contiguous memory-write operations to the DRAM controller, and consequently degrade the controller's performance owing to DRAM row buffer conflicts [33]. We observe the phenomenon by using a detailed memory controller from a commercial CPU rather than a fixed-cycle DRAM simulator.

The rest of the paper is organized as follows. In Section 2, we revisit the DMA mechanism and discuss the characteristics of DMA memory reference. Section 3 presents the design of DDC and PBDC. In Section 4, we describe the experimental setups. Section 5 presents and discusses the experimental results. Section 6 and Section 7 present related work and summary respectively.

2. Revisiting DMA Mechanism

Before revisiting the DMA mechanism, we introduce several important terms:

I/O data and CPU data: we define I/O data as a piece of data produced by I/O devices directly. The data retains the "I/O_attribution" until it is freed or updated by the CPU. CPU data is defined as two parts: 1) the data not produced by the I/O devices and 2) the I/O data updated by the CPU. Take receiving network packets as an example, the network DMA engine performs the DMA-write operations to write payloads into the buffer "sk_buffer->data", wherein the data is considered as I/O data until it is freed. When the data is copied from the "sk_buffer->data" to other places (e.g., user buffer) by the CPU executing data movement instructions, the new data in the destination buffer is CPU data. If CPU updates the data in the "sk_buffer->data", the data also turns to be CPU data. Note that the data retains its attribution when transferred among caches via the interconnect network or system bus in a multicore/multi-processor system.

DMA direction: there are two directions for DMA operations, i.e., "DMA-Produce-CPU-Consume (DPCC)" and "CPU-Produce-DMA-Consume (CPDC)". The DPCC direction indicates the DMA write operations that write I/O data into the memory and the CPDC direction indicates the DMA read operations that read I/O data from the memory.

DMA request: usually one DMA request consists of a number of memory references, each of which is for one cache line. DMA request size is the memory footprint size of one DMA request (see "DMA req size" in Figure 2).

Reuse Distance: the reuse distance (also called LRU stack distance [17]) reveals data's locality. As illustrated in Figure 2, the reuse distance is the number of distinct data elements, which are in terms of cache line in this

paper, accessed between two consecutive references to the same element. The I/O data reuse distance in the DPCC direction indicates the number of the distinct cache lines accessed between when the data is produced by the I/O devices and when it is consumed by the CPU, or vice versa.

2.1 The Details of the DMA Operation

We take one DMA direction, i.e., the "DMA-Produce-CPU-Consume (DPCC)" direction, as an example to present the detailed processes of the DMA operations. Figure 1 illustrates the interactions of processor, memory and DMA engine (I/O device) in the DMA operations. The interactions require three data structures, i.e., DMA buffer, descriptor and destination buffer. Usually DMA buffer's start address varies in a physical address space and the DMA buffer's size also varies. To manage these discrete DMA buffers, the driver uses the DMA descriptors, each of which includes a pointer to the DMA buffer's start address and a variable of the DMA buffer's size, and several variables for status information such as the DMA buffer's owner (could be the processor or the DMA engine). The detailed processes of the DMA receiving operation, i.e., the "DPCC" direction, are described as follows (see Figure 1):

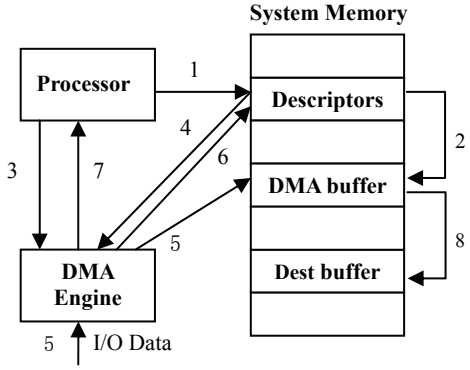

Figure 1 The Interactions of processor, memory and DMA engine in DMA operation

1). The device driver (processor side) creates a descriptor for a DMA buffer. 2). The driver allocates a DMA buffer in the memory and initializes the descriptor with the DMA buffer's start address, size and status information. 3). The driver informs the DMA engine of the descriptor's start address. 4). The DMA engine loads the descriptor's content from the memory. 5). With the DMA buffer's start address and size information extracted from the descriptor, the DMA engine receives the data from the I/O device and writes the data to the DMA buffer. 6). After all I/O data is stored in the DMA buffer, the owner status of the descriptor is modified to be the DMA engine. 7). The DMA engine sends an interrupt to the processor to indicate the completion of the receiving operation. 8). The driver handles the interrupt raised by the DMA engine and copies the received I/O data from the DMA buffer to the Destination buffer. Then, it frees the DMA buffer. After all is done, go to 2).

Figure 2, from an architectural point of view, illustrates the DMA data transfer flow in the DPCC direction on the modern mainstream machines wherein the processors adopt snooping-cache scheme for maintaining I/O data's coherence so that they need to send the snoop requests to the processor's data cache to invalidate those cache blocks that the I/O data is hit. Consequently, when the CPU consumes the I/O data, the compulsory misses will take place and trigger the memory read requests to the memory controller. (The data transfer flow for network I/O can be found in the recent publications [18]).

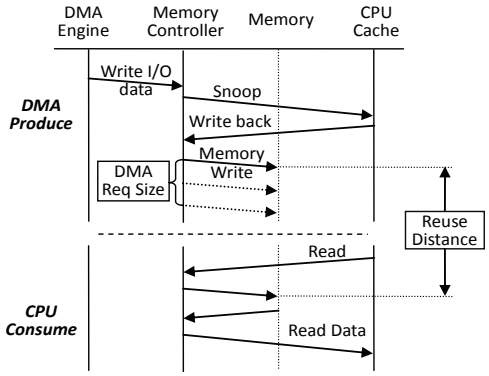

Figure 2 DMA data transfer flow in the "DMA-Producer-CPU-Consumer (DPCC)" direction

2.2 The Characteristics of DMA Memory Reference

We investigate the inherent characteristics of the DMA memory reference, including the distributions of various I/O types, the DMA request size, and the reuse distance. We select three typical applications, File-Copy (a 400MB file), SPECWeb2005 and TPC-H, because their behaviors can represent the most other I/O intensive applications we have examined. (More details on the experimental setups and the trace collection are in Section 4).

Table 1 shows that the percentages of the DMA memory references vary largely. For File-Copy application, about 40% of the memory references are relative to DMA operations; moreover the DMA read references (19.6%) are almost equal to the DMA write references (19.3%). For TPC-H dominated by the DMA write requests (i.e., read data from disk to memory), although the percentage of all DMA memory references is about 20%, the DMA write references account for 19.9%. For SPECweb2005, however, the percentage of DMA memory references is only 1.0%. We find that the average size of the network I/O requests is so small (< 0.3KB) that the processor is often busy with handling interrupts.

Table 2 and Figure 3 show that the DMA request sizes depend on both the I/O types and the application's behavior. In the file-copy and TPC-H applications, the size of all DMA write requests is less than 256KB and the percentage of the requests with the size of 128KB is about 76%. For the two applications, the average sizes of the DMA write requests are about 110KB and 121KB, respectively. For SPECweb2005, the size of all NIC DMA requests is smaller than 1.5KB because the maximum transmission unit (MTU) of the Gigabit Ethernet frame is only 1518 bytes; compared with File-Copy and TPC-H, the size of the IDE DMA requests is also very small, an average of about 10KB.

The fifth column of Table 2 presents the percentages of the sequential DMA memory references (i.e., Ref_n =

Ref_{n-1} + 1), which can be exploited by numerous optimization approaches such as prefetching for improving application's performance. According to the table, most of the DMA memory references have more regular patterns than the CPU memory references. Take TPC-H as an example: within one IDE DMA write request whose average size is 119KB, about 96.8% of the DMA memory references are sequential, covering the 115KB memory footprint. In contrast, only 4.1% of the CPU write references are sequential. Nevertheless, the NIC DMA write references, as an exception, have very poor regularity, because the payloads are so small that they usually cover only one or two cache lines. Ordinarily, the larger the DMA request size, the more regularity there is within the DMA request.

Table 1 Percentage of Memory Reference Types

	File Copy	TPC-H	SPECWEB
CPU Read	33.4%	60%	75%
CPU Write	27.7%	20%	24%
DMA Read	19.6%	0.1%	0.76%
DMA Write	19.3%	19.9%	0.23%

Table 2 Average Size of Various Types of DMA Requests

	Request Type	Type (%)	Avg. Size	SeqRef(%)
File Copy	IDE DMA Read	50.5%	393KB	98.9%
	IDE DMA Write	49.5%	110KB	96.7%
	CPU Mem Read	-	-	58.5%
	CPU Mem Write	-	-	60.6%
TPC-H	IDE DMA Read	0.5%	19KB	73.2%
	IDE DMA Write	99.5%	121KB	96.8%
	CPU Mem Read	-	-	45.2%
	CPU Mem Write	-	-	4.1%
SPECWEB	IDE DMA Read	24.4%	10KB	95.3%
	IDE DMA Write	1.7%	7KB	93.9%
	NIC DMA Read	52%	0.3KB	48.6%
	NIC DMA Write	21.9%	0.16KB	4.1%
	CPU Mem Read	-	-	25.5%
	CPU Mem Write	-	-	1.0%

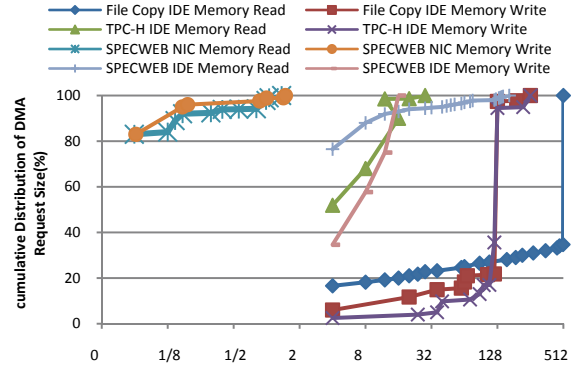

Figure 3 Cumulative distribution of DMA Request Size (KB)

Figures 4~6 illustrate the reuse distance statistics of the I/O data and the CPU data. Comparing Figure 4 with Figure 6, the I/O data's reuse distances in the DPCC direction are extremely smaller than the CPU data's reuse distance; but from Figure 5, the I/O data's reuse distance in the CPDC direction are very large. In the DPCC direction, the portions of the small reuse distances ($\leqslant$ 32K cache lines) are near 90% for the disk I/O data and about 48% for the network I/O data. Note that, 32K cache lines correspond to 2MB, for one cache line is 64-Byte. The small reuse distances indicate that the read latency is significant to applications so that OS drivers tend to process the I/O data immediately after the interrupt of a DMA write operation completion. In contrast, the write latency is unimportant, thus the I/O write requests can be buffered and combined with other write requests. In the CPDC direction, the reuse distances of the disk I/O data in the file-copy and TPC-H applications are larger than 128K cache lines (8MB). The reuse distances of the disk I/O data in SPECWeb is about 8K~128K cache lines (512KB~8MB), accounting for a portion of over 80%. The network I/O is different from the other types of I/O, because the latencies of both transmitting and receiving operations are crucial. Therefore, in the CPDC direction, the reuse distances of the network I/O data (i.e., "SPECWeb CPU-W-NIC-R" in Figure 5) are also less than 8K cache lines (512KB).

2.3 Architecturally Separating I/O data from CPU data

As discussed in the last subsection, the characteristics of DMA memory references to I/O data are indeed different from the characteristics of memory references to CPU data. The differences include: (1) the memory references to the I/O data have more regularity, but the reference patterns to the CPU data are uncertain; (2) the I/O data's reuse distances in the DPCC direction are smaller than the CPU data's reuse distances; (3) usually the I/O data are used only once, and they are freed after being copied, but the CPU data might be reused many times. Therefore, separating the I/O data from the CPU data is more likely to leverage the inherent characteristics of DMA memory reference for improving I/O performance than unifying them in one cache.

Although it is easy to conceptually separate I/O data from CPU data (as defined in Section 2), it is difficult to fully separate them on the architectural perspective. There are two general methods to architecturally separating data. The first method, using dedicated on-chip storage and data path, is widely used for instruction cache and data cache in the contemporary processor. Unlike the instructions which are essential for program execution, I/O operations are mainly performed in the I/O-intensive applications. Therefore, using dedicated on-chip storage for the I/O data can raise an important cost problem. The second method is using the tainting scheme that adopts one bit to tag data. Tagged prefetching [32] uses this method to identify prefetched data. In recent years, the tainting method is widely studied for software security [25] by tagging the data as untainted (safe) and tainted (unsafe). Nevertheless, many hardware tainting schemes require a redesign of the entire computer system including ALUs, registers, caches, buses and memory etc.

In this paper, we propose a DMA cache technique to architecturally separate I/O data from CPU data by combining the two methods. In the last level of the cache hierarchy, the DMA cache technique adopts a dedicated on-chip storage to store the I/O data; at the low levels (i.e., L1 and L2 caches), it uses one bit to tag a cache block to

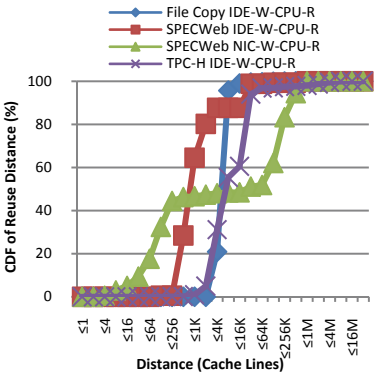

Figure 4 Cumulative distribution (CDF) of reuse distance of I/O data in the DPCC direction

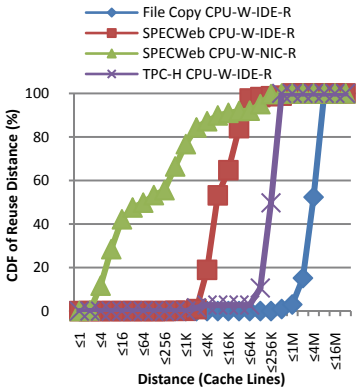

Figure 5 Cumulative distribution (CDF) of reuse distance of I/O data in the CPDC direction

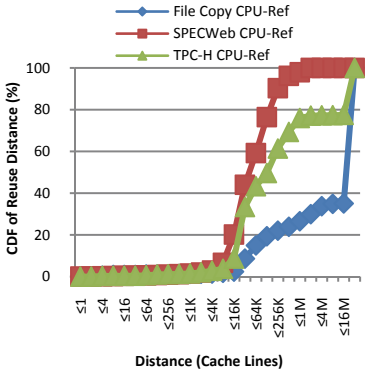

Figure 6 Cumulative distribution (CDF) of reuse distance of CPU data

indicate whether the block stores the I/O data or the CPU data. Note that the DMA cache technique is not a fully architectural separation method because the memory is unified. Therefore, if the I/O data is written back from the dedicated DMA cache to the memory, it would convert into CPU data. In other words, later the data will be refilled into the processor's cache rather than the DMA cache. We will discuss the design of the DMA cache in detail in the next section.

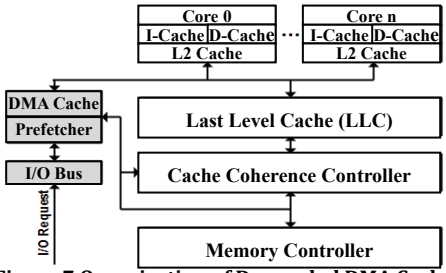

Figure 7 Organization of Decoupled DMA Cache

3. Two Designs of the DMA Cache

Based on the concept of the DMA cache, we present two concrete DMA cache designs in this section. The first, Decoupled DMA Cache (DDC), adopts dedicated on-chip storage as the DMA cache to buffer I/O data and is suitable for I/O-specific processors. The second design, Partition-Based DMA Cache (PBDC), can dynamically use some ways of a processor's LLC as the DMA cache and is suitable for general purpose processors.

3.1 Decoupled DMA Cache (DDC)

3.1.1 Overview of DDC

As illustrated in Figure 7, the core of DDC consists of a dedicated DMA Cache, Prefetcher and control/data paths. The DMA Cache stores only I/O data and the LLC stores only CPU data. Prefetcher is responsible for fetching I/O data into DMA Cache in the CPDC direction. In the DDC design, the cache coherence controller (CC-Ctrler) and the LLC are not modified, reducing design complexity, but the L1/L2 cache require adding one bit to each block tag in order to indicate whether the block stores I/O data or CPU data.

3.1.2 Write Policy of DMA Cache

Because cache write policy can influence coherency protocol and replacement policy, we first discuss it. There are two write policies, i.e., write-back (WB) and write-through (WT). According to Subsection 2.2, because the characteristics of DMA memory references to I/O data are different from the characteristics of memory references to CPU data, it might be unreasonable for the DMA Cache to simply adopt the same write policy of the processor's data cache. Therefore, we desire to investigate both the WB and WT policies. When adopting WB policy, the I/O data written to the DMA Cache is initiated as Modified-state and deferred to be updated into the memory until it is replaced. When adopting WT policy, the I/O data is written to the DMA Cache as well as the memory; its initial coherence state is Exclusive-state so that the I/O data can be replaced by later DMA writes directly without extra update operations.

3.1.3 Cache Coherence

To address the cache coherence challenge, we have refined the MOESI protocol [3] and the ESI protocol [16] for WB and WT policies respectively, i.e., the IO-MOESI (Figure 8) and the IO-ESI (Figure 9) protocols. Essentially, the refined cache coherence protocols for the DMA Cache are the same as the original MOESI/ESI protocols.

The only difference between the IO-MOESI/IO-ESI and the original protocols is exchanging the local sources and the probe sources of the state transitions (see Table 3). For example, if a CPU generates a write request to its cache (i.e., local cache), the other CPU's caches (i.e., remote cache) and the DMA Cache might receive the same "Probe-Write" request. Therefore, the DMA Cache can be viewed as a remote cache.

For the DMA Cache, when using WB policy, it can use the IO-MOESI protocol. When using WT policy, it can use the IO-ESI protocol, in which if a read miss or a write miss occur in the DMA Cache, the initial coherence state for the filled data is Exclusive-state rather than the Modified-state in MOESI protocol.

According to Figures 8~9, we describe the three main DMA Cache transactions to show how the protocols work (please refer to [31] for more details).

Table 3 Sources of State Transitions

	Local Cache	Remote Cache	DMA Cache
	MOESI/ESI	MOESI/ESI	IO-MOESI/IO-ESI
CPU Read Request	CPU-Read	Probe-Read	Probe-Read
CPU Write Request	CPU-Write	Probe-Write	Probe-Write
DMA Read Request	Probe-Read	Probe-Read	DMA-Read
DMA Write Request	Probe-Write	Probe-Write	DMA-Write

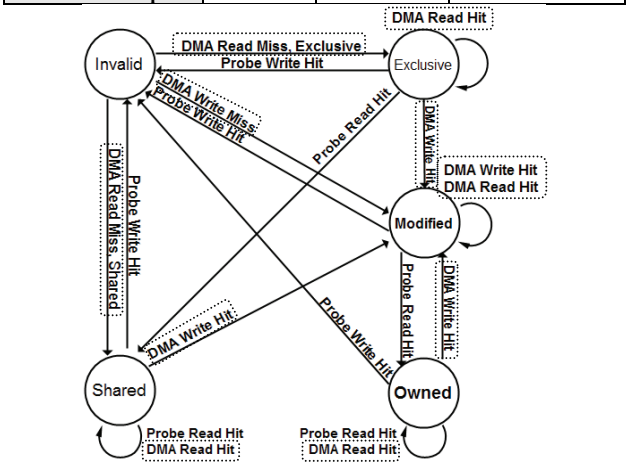

Figure 8 IO-MOESI State Transitions for DMA Cache with Write-Back Policy[1]

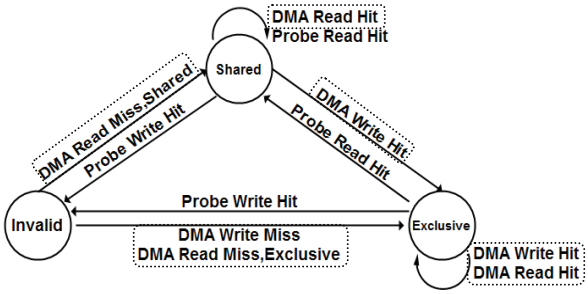

Figure 9 IO-ESI State Transitions for DMA Cache with Write-Through Policy[2]

(1) When the DMA Cache receives a DMA write request, it first broadcasts a Probe-Read request to other caches. If the Probe-Read request hits in a CPU's LLC, the LLC changes the hit block's state to either Shared-state or Owned-state, and sends data to the DMA Cache. After receiving the most recent data, the DMA Cache broadcasts a Probe-Write. The LLCs, in which the Probe-Write hit, change the hit block with Shared-state or Owned-state to Invalid-state. Then, the DMA Cache changes the block to Modified-state (using IO-MOESI with WB policy) or Exclusive-state (using IO-ESI with WT policy where the data is updated into memory as well). Then, a DMA write operation is completed.

(2) When a LLC meets read-miss, it broadcasts a Probe-Read request. If the most recent data is in the DMA Cache, the data would be refilled to the L2 cache directly, bypassing the LLC. The DMA Cache changes the hit block's state from Modified-state to Owned-state (using IO-MOESI with WB policy), or from Exclusive-state to Shared-state (using IO-ESI with WT policy). The data refilled into L2 cache is Shared-state and tagged with "IO_Data" attribution. Then, the CPU read operation is completed. Note that the block with "IO_Data" tag in the L1/L2 cache is evicted directly without being swapped into the victim buffer or the LLC.

(3) When a DMA read request arrives at DMA Cache, if a miss occurs, a Probe-Read request is broadcast to other caches via CC-Ctrler. If the request does not hit in any caches, it is forwarded to the memory controller. After receiving the desired data from the memory, the DMA Cache sets the data as Exclusive-state; if a copy of the data with Modified-state exists in other processor's cache, the processor's cache forwards the data to the DMA Cache and changes the block's state to Owned-state, and the DMA Cache set the block as Shared-state. Then, the DMA read operation is completed. Note that a DMA read operation can trigger prefetching operations that will be described in Subsection 3.1.5.

3.1.4 Replacement Policy of DMA Cache

Because usually I/O data is used only once, it first evicts the used I/O data. When using the IO-MOESI protocol, the DMA Cache first selects the blocks with Invalid-state, Shared-state and Owned-state for replacement and then selects the Exclusive-state blocks; finally the Modified-state blocks are selected. When using the IO-ESI protocol, the DMA Cache first selects the blocks with Invalid-state and Shared-state and then the Exclusive-state blocks. If two or more blocks with a same state exist for replacement, the LRU policy is used.

3.1.5 Other Design Issues

In the CPDC direction, the DMA read request on the DMA Cache can trigger a prefetching operation. As mentioned in Subsection 2.2, the I/O data has very regular reference patterns, most being linear within a DMA buffer. Therefore, the DMA Cache could benefit from adopting even a straightforward sequential prefetching technique. In our design, the prefetcher is triggered to generate four sequential prefetching requests upon the state transitions from Invalid-state to Shared-state or Exclusive-state. The prefetching requests are forwarded to the cache coherence controller as a Probe-Read request and the desired data can be fetched from either some processor's caches or the memory of the system.

The L1/L2 caches add one bit to each block tag to indicate whether the block stores I/O data or CPU data. The tag of the block can be propagated among the L1/L2 caches. The replacement policy of the L1/L2 caches requires a slight modification in which the block with the "IO_Data" tag is evicted directly without being swapped into the victim buffer or the LLC. To propagate the tag, the data paths also need to add one bit. When forwarding blocks to other caches, the DMA Cache and LLC tag the data copy as "IO_Data" and "CPU_Data", respectively. In other words, the DMA Cache is the source of "IO_Data" tag and the LLC is the source of "CPU_Data" tag.

3.2 Partition-Based DMA Cache (PBDC)

3.2.1 Overview of PBDC

Partition-Based DMA Cache (PBDC) is designed to achieve the same effect as DDC but without additional on-chip storage. From Figure 10 that illustrates the organization of PBDC in a multiprocessor system, PBDC

[1,2] The dashed boxes indicate the changes to the original MOESI/ESI, i.e., replacing "CPU-Read/CPU-Write" with "DMA-Read/DMA-Write".

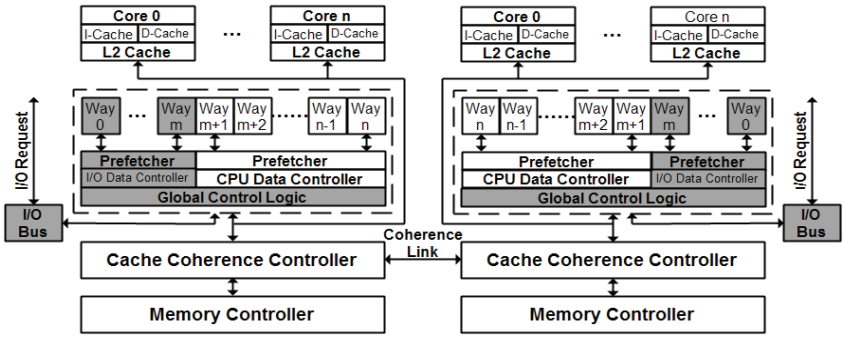

Figure 10 Organization of Partition-Based DMA Cache (PBDC)

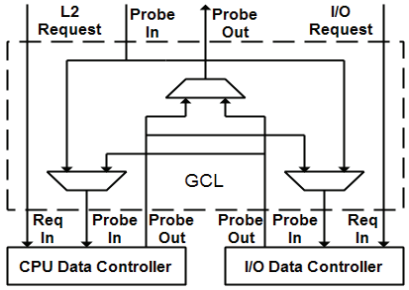

Figure 11 Global Control Logic (GCL) Delivering Requests

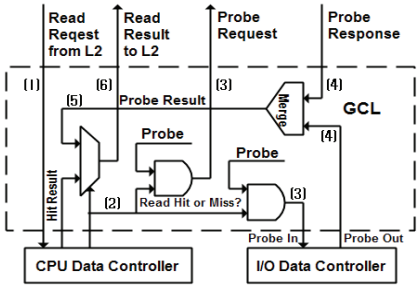

Figure 12 The Process Flow of L2 Cache Read Request

can use several ways of the processor's LLC as the dedicated DMA Cache. Because some important design issues of PBDC, such as write policy, cache coherence protocol, replacement policy, can adopt the same design of DDC, we mainly focus on the design of the LLC's controller.

3.2.2 The Design of Last Level Cache Controller (LLC-Ctrler)

The LLC-Ctrler consists of Global Control Logic (GCL), IO Data Controller (IOD-Ctrler), CPU Data Controller (CPUD-Ctrler), two Prefetchers and Configuration Module (CM).

Global Control Logic (GCL) is responsible for routing the read/write requests and the Probe-Read/Probe-Write requests to the right destinations. As shown in Figure 11, it can 1) receive read/write requests from L2 caches and forward them to CPUD-Ctrler, 2) receive the read/write requests from the I/O bus and forward them to IOD-Ctrler, 3) receive the Probe-Read/Probe-Write requests from CC-Ctrler and forward them to both IOD-Ctrler and CPUD-Ctrler, 4) receive the probe requests from IOD-Ctrler and forward them to CPUD-Ctrler and CC-Ctrler, and 5) receive the probe requests from CPUD-Ctrler and forward them to IOD-Ctrler and CC-Ctrler.

I/O Data Controller (IOD-Ctrler) and CPU Data Controller (CPUD-Ctrler) are responsible for write policy, maintaining cache coherence, replacement policy and tagging data for the I/O data ways (Way_0 ~ Way_m) and the CPU data ways (Way_{m+1} ~ Way_n) respectively. CPUD-Ctrler still adopts the original cache management policies. IOD-Ctrler can use different policies for the I/O data, e.g., the WT policy, the IO-ESI protocol and the replacement policy of the DDC.

Figure 12 illustrates the flow of an L2 read request. (1) When GCL receives the read request from the L2 cache, it forwards the request to CPUD-Ctrler. (2) If the request hits in the CPU data ways, the block tagged with "CPU_data" is refilled to the L2 cache directly. (3) Otherwise, a probe-read request is triggered and forwarded to CC-Ctrler. Meanwhile, an internal probe-read request is also generated to IOD-Ctrler. (4)~(5) GCL merges all probe responses and (6) sends the required data to the L2 cache. If the data is from the IO data ways (Way_0 ~ Way_m), it would be tagged as "IO_data" and IOD-Ctrler sets the block state as Owned-state or Shared-state. Then, the "L2 cache miss read request" is completed.

Configuration Module (CM) consists of a register. The register has the same bit-width as the number of cache ways and each bit indicates one way. If the bit is set to "1", the corresponding way is for I/O data; otherwise, the way is for CPU data. Upon each read/write request, IOD-Ctrler and CPUD-Ctrler use the register as the mask to lookup the cache tags of the n-ways. Note that the register can be dynamically configured by BIOS or OS drivers but then the flush operations to the corresponding ways are required.

3.3 Discussion of Design Complexity and Cost

There are trade-offs of design complexity and design cost for the two designs. On design complexity, although the designs both require the modifications of write policy, cache coherence protocol, replacement policy and prefetcher, the design of DDC mainly focuses on the the additional DMA Cache, but PBDC requires substantial modifications of the processor's LLC controller. Therefore, DDC has less complexity than PBDC. On design cost, because PBDC requires no additional on-chip storage and can dynamically use several LLC's ways as the DMA cache, it has less cost than DDC.

In summary, DDC is suitable for the I/O-specific

processors such as embedded processors, which are designed for a class of the applications with known program characteristics. PBDC is suitable for general purpose processor because it is more flexible and has less cost. For example, in a multiprocessor platform, we can enable the DMA Cache function of only those processors connected with I/O devices and disable others. Furthermore, we can configure large DMA Caches for I/O-intensive applications and even disable them for computing-intensive applications.

4. Experimental Setup

4.1 Applications

We select three typical applications, because their behaviors can represent most of the other I/O intensive applications that we have examined. The selected applications are described as follows:

(1) File-copy: A real-world application that copies one 400MB file (SPECCpu2006 install package) within Ext3 file system on the Linux platform. **(2) TPC-H** [7]: A decision support benchmark. It consists of a suite of queries and concurrent data modifications. The queries and the data reside in an Oracle 11g database with 10GB data. **(3) SPECweb2005** [6]: A SPEC benchmark for evaluating the performance of web servers. It is used for measuring a system's ability to act as a web server.

4.2 The Trace Collection and the FPGA Emulation Platform

Table 4 Evaluation Parameters of CPU Last Level Cache and DMA Cache

	CPU LLC Parameters	DMA Cache Parameters
Size	2MB/4MB/8MB	512K/256K/ 128K/64K/32K
Cache Line	64Byte	64Byte
Associative	8Way/16Way	1Way/2Way/4Way
Replace Policy	Random	See Subsection 3.1.4
Write Policy	CPU Data Write Allocate & Write Back (WB)	1. Write Allocate & Write Back 2. Write Allocate & Write Through
	I/O Data 1. Write Allocate & Write Back (WB) 2. Write Allocate & Write Through (WT)	
Access Time	20 Cycles	20 Cycles

Note: The WT policy for I/O data updates both the cache and the memory. If a cache block is partially modified, the fetch-on-write operation is triggered (i.e., the block will be read from the memory before being updated). Our studies indicate that the probability of the partially modified I/O write is very small.

We run all applications on an AMD Opteron 2.2GHz sever with 2GB dual-channel DDR memory. We use the HMTT system [10], a tool plugged in an idle DIMM, to collect all memory reference traces. Moreover, the HMTT system is capable of distinguishing whether a memory reference is issued by the DMA engine or the processor. Using the HMTT system, we have collected 40M memory references (requests) for each application. Because each requests is for 64 bytes (one cache line), the footprint of 40M memory requests is about 2.6GB. It should be reasonable for studying the behaviors of DMA data transfer.

We have implemented our two designs as well as the shared-cache and Prefetch Hint [18] schemes in an FPGA emulation platform. Our cycle-accurate emulation platform consists of a last level cache and a DDR2 controller from Godson-3 multicore processor [13], a DDR2 DIMM model from Micron Technology. All caches work at a frequency of 2GHz and the DDR2 controller works at 333MHz (i.e., DDR2-666). The whole emulation platform is implemented in synthesizable RTL codes. We use an FPGA based RTL emulation accelerator, the Xtreme system [8] from Cadence, to accelerate the emulation platform. More detailed emulation parameters are listed in Table 4.

5. Experimental Results

In this section, we first evaluate the effect of separating I/O data and CPU data (in section 5.1), the impact of cache write policy (in section 5.2) and the cache configurations (in section 5.3) by analyzing DDC's performance, and then we will compare the performance of DDC, PBDC and the other schemes in section 5.4.

5.1 Separating I/O data and CPU data vs. Unifying them

Figure 13 illustrates the normalized speedup of the shared-cache scheme and DDC of various sizes and WB policy (baseline is the memory cycles spending on the memory bus in the existing snooping-cache scheme). We find that even the small-size DMA caches can outperform the shared cache scheme. For the DDC with Prefetcher, the performance improvements are 30.1%, 11.9% and 10.2% for the three applications when setting the DMA cache size to 256KB. However, the performance of DDC is sensitive to the cache size. For example, the 64KB/32KB DMA caches may exhibit even worse performance than the baseline for file-copy and TPC-H applications. To find out the reason, we can review Figure 3 which shows that over 80% of the DMA requests' sizes are larger than 64KB in the two applications. This characteristic causes frequent replacements for the modified blocks in the small-size DMA caches. Figure 17 illustrates that the replaced modified blocks account for over 90% and 60% for file-copy and TPC-H respectively. Unfortunately, those replaced modified blocks are non-contiguous, causing poor DDR memory performance due to lots of row buffer conflicts [33]. In SPECweb2005, this phenomenon disappears because the average size of the DMA requests is less than 10KB (see Table 2).

Nevertheless, the small DMA request size can cause the most CPU references to the I/O data being not sequential, which can also cause poor DRAM performance due to the row conflicts. When the CPU references the I/O data from the DMA cache, the "CPU RD MEM" cycles (see Figure 18) are significantly

reduced. Therefore, even though the percentage of the DMA memory references in SPECWeb is small, the optimization effect of the DMA cache is still obvious.

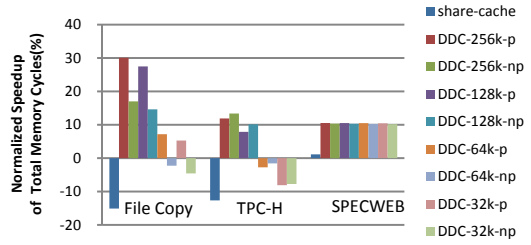

Figure 13 Normalized Speedup with Write Back Policy

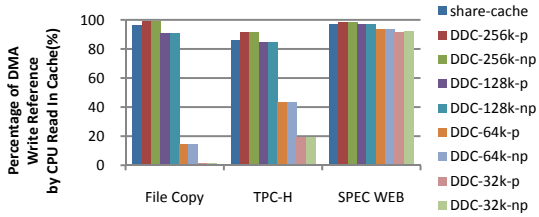

Figure 14 Cache Hit Rate for I/O Data

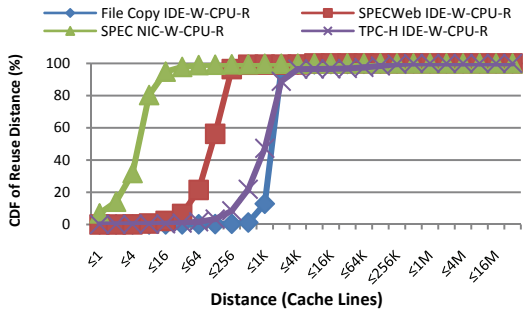

Figure 15 Cache Hit Rate for I/O Data

Further, we investigate the "Cache Hit Rate for I/O Data" metrics shown in Figure 14. The hit rate of the share-cache for I/O data is determined by the reuse distance of the I/O data (Figure 4) and the hit rate of DMA cache is determined by the number of the distinct I/O data cache lines between the I/O write reference and the CPU read reference (Figure 15). Figure 4 shows that the 90th percentile of the produce-consume reuse distances is around "⩽32K" cache lines (i.e. 2MB). Figure 15, however, indicates that if separating I/O data from CPU data, the produce-consume distances are significantly reduced, all being less than 4K cache lines (i.e., 256KB). Take "TPC-H IDE-W-CPU-R" as an example, 96.4% of the reuse distance is less than 4K cache lines (256KB) and 47.2% are less than 1K cache lines (64KB). Consequently, the hit rate of the DMA cache also changes down. It is interesting that although the shared-cache decreases the performance for file-copy (-15.1%) and TPC-H (-12.7%), it exhibits higher cache hit rate for the I/O data than the 128KB DMA cache without Prefetcher. We find that the reason is related to the write-back policy and will present the reason in more detail in next subsection.

5.2 Write-Back Policy vs. Write-Through Policy

When using the write-through policy, according to Figure 16, all schemes including the shared-cache and the small-size DMA caches can achieve improvements. The DDC, with Prefetcher and WT, can achieve the speedups of 58.4%, 35.4% and 10.5% when setting the DMA cache size to 256KB. Moreover, in contrast to the negative speedups (-15.1%, -12.7% and 1.1%) of the shared-cache with WB policy shown in Figure 13, the shared-cache with WT policy can even achieve positive speedups by 15.3%, 17.1% and 1.2%.

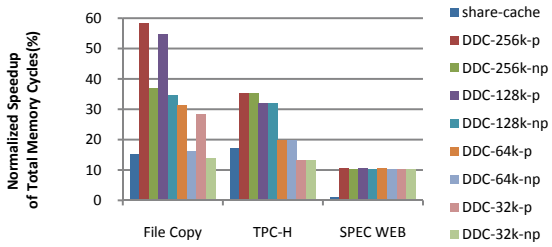

Figure 16 Normalized Speedup with Write Through Policy

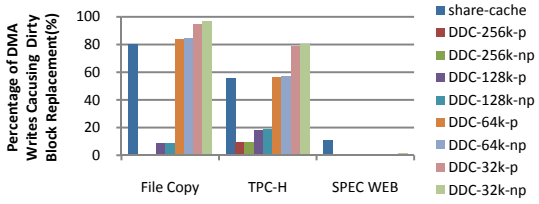

Figure 17 The Percentage of DMA Writes Causing Modified Block Replacement

We investigate why the write policy has a substantial impact on the performance and find that the portion of replaced modified blocks accounts for 80.0% and 55.8% for shared-cache in the file-copy and TPC-H application respectively (see Figure 17). Unfortunately, because the replaced modified blocks can cause non-contiguous write-back requests, the DDR DRAM exhibits poor performance due to lots of row buffer conflicts. Moreover, the contiguous I/O data may also be written back to memory non-contiguously when they are replaced later. Therefore, the memory access overhead due to the DMA write requests is substantial. For example, according to Figure 18, the portion of the "DMA WR MEM" in shared-cache scheme accounts for 31.9% in file copy application. The larger the DMA request size, the worse is the negative influence on the DRAM system. Therefore, because the average size of the DMA requests in SPECWeb is less than 10KB, the negative influence on the DRAM system is not obvious. It should be noted that this phenomenon would be hidden in a fixed-cycle DRAM simulator. We reveal it by using a detailed memory controller from a commercial CPU.

When using the write-through policy, the I/O data is written into the cache as well as the memory. Because the I/O data in the DMA cache is with Exclusive-state, they can be invalidated without any additional operations. Consequently, the I/O data can be written in the memory

contiguously to reduce the substantial row buffer conflicts. Take file-copy as an example, the portion of "DMA WR MEM" cycles (see Figure 19) of the DMA cache with WT schemes is reduced from 15.9% to 3.0%, which is close to the portion in the snooping-cache scheme (2.7%). For the shared-cache scheme, this portion is also reduced significantly, from 31.9% to 8.6%, but it is still three times more than the snooping-cache scheme because of cache interferences between the CPU data and the I/O data. The interferences can be eliminated in the DMA cache schemes. Thus, from Figure 16, even the 128KB DDC can outperform the shared caches by 2.9X (32.5%/11.2%) on average for all applications. Note that we will adopt the WT policy to evaluate the following experiments.

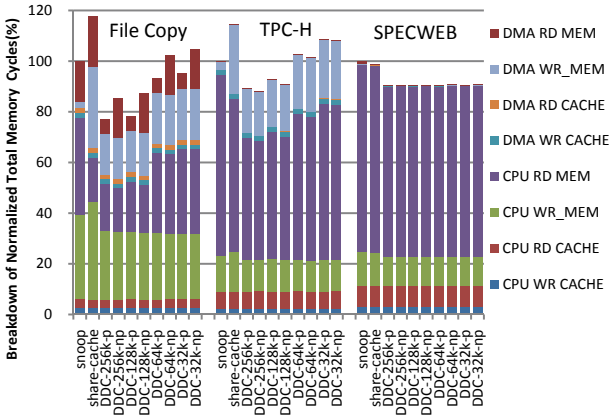

Figure 18 Breakdown of Normalized Total Memory Cycles with Write-Back Policy

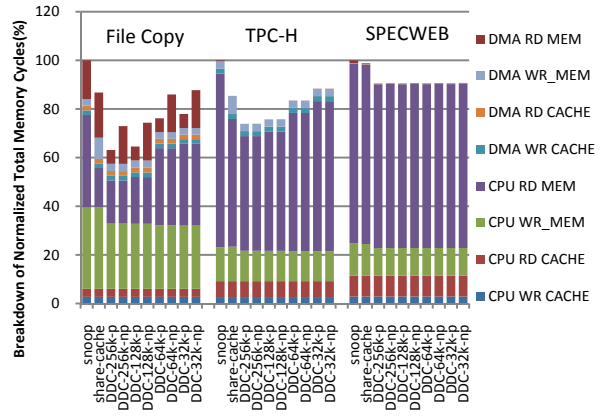

Figure 19 Breakdown of Normalized Total Memory Cycles with Write-Through Policy

5.3 DMA Cache's Associativity and Size vs. Processor LLC Size

We fix the processor's LLC size to be 2M and change the DMA cache's associativity from 1-way to 4-way. Table 5 shows that the associativity parameter has little impact on the performance of the schemes. For example, when the DMA cache's associativity changes from 1-way to 4-way, the performance of the 256KB DMA cache with the prefetcher is only improved by about 1.3%, 0.2%, 0.1%. The DMA cache size, however, has significant impact on the performance. Take the 1-way DMA caches with prefetching for example, when the sizes change from 256KB to 32KB, the improvements are substantially decreased by 28.6% and 22.1% in file-copy and TPC-H applications.

We change the processor LLC size from 2MB to 8MB and fix the DMA cache's associativity to be 4-way. Table 6 shows that the processor's LLC size has a considerable impact on the performance of the schemes. For example, when the processor LLC size changes from 2MB to 8MB, the performance of the 256KB DMA cache with the prefetcher is improved by 5.5%, 11.3% and 5.4% for the three applications respectively. Actually, the 128KB DMA cache with a 2MB-LLC can substantially outperform the share cache with a configuration of an 8M-LLC and the WT policy.

Table 5 Normalized Speedup (%) of Different DMA Cache Associative

	DMA Cache Associative	DMA Cache (2MB Processor LLC)							
		256KB		128KB		64KB		32KB	
		P	NP	P	NP	P	NP	P	NP
File Copy	1 Way	57.0	36.0	50.1	30.1	30.1	16.5	28.2	13.8
	2 Way	57.9	36.7	50.5	31.1	31.6	16.6	28.3	13.9
	4 Way	58.3	37.0	54.9	34.5	31.3	16.3	28.3	13.9
TPC-H	1 Way	35.2	35.2	31.6	31.5	19.5	19.5	13.1	13.1
	2 Way	35.3	35.2	31.8	31.8	19.7	19.8	13.2	13.2
	4 Way	35.4	35.3	32.1	32.0	19.8	19.9	13.2	13.2
SPEC Web	1 Way	10.4	10.2	10.3	10.1	10.4	10.1	10.3	10.1
	2 Way	10.5	10.3	10.4	10.2	10.4	10.2	10.4	10.2
	4 Way	10.5	10.3	10.5	10.2	10.4	10.2	10.4	10.2

Table 6 Normalized Speedup (%) of Different CPU LLC Size

	CPU Cache Size	Shared Cache	DMA Cache (4-Way)							
			256KB		128KB		64KB		32KB	
			P	NP	P	NP	P	NP	P	NP
File Copy	2M	15.2	58.3	37.0	54.9	34.5	31.3	16.3	28.3	13.9
	4M	16.1	61.4	38.7	57.7	36.0	32.8	17.1	29.6	14.6
	8M	16.9	63.8	40.0	57.6	37.2	34.0	17.6	30.7	15.1
TPC-H	2M	17.1	35.3	35.3	32.0	32.0	19.8	19.8	13.2	13.2
	4M	20.3	40.5	40.4	36.4	36.4	21.8	21.8	14.1	14.1
	8M	24.3	46.6	46.5	41.4	41.4	23.6	23.5	14.4	14.4
SPEC Web	2M	1.1	10.5	10.3	10.5	10.3	10.4	10.2	10.4	10.1
	4M	1.7	13.0	12.8	12.9	12.7	12.9	12.7	12.9	12.7
	8M	2.6	15.9	15.7	15.8	15.7	15.7	15.6	15.7	15.5

As mentioned in Section 3.1.5, we have integrated a sequential prefetching with a prefetching degree of four cache blocks into the DMA cache schemes. The two tables also show the effect of the prefetching for the DMA read requests. Take file-copy that has a large portion of DMA read request (see Table 1) for example: according to Table 6, the 256KB DMA cache schemes with prefetching outperform the non-prefetching schemes by over 20%.

5.4 Decoupled DMA Cache vs. Partition-Based DMA Cache

We compare DDC and PBDC as well as the shared-cache and Prefetch Hint [18] in this subsection. In the PBDC scheme, we have evaluated two LLC configurations, i.e., using one way and two ways as the

DMA cache respectively. Note that, in a 2MB 16-way set-associative LLC, one way is 128KB; and in a 4MB 16-way set-associative LLC, one way is 256KB.

According to Figures 20 and 21, we can find that: (1) DDC demonstrates better improvements than PBDC. For example, with a 4MB LLC, the 256KB (1-way) PBDC can achieve the improvements by 50.4%, 31.7% and 5.2%, and the speedups of the 256KB DDC are 61.7%, 40.5% and 13.0%. Therefore, **with less design cost, PBDC can achieve nearly 80% of the improvements of DDC**; (2) the shared-cache with WT scheme shows the performance improvements by 16.1%, 20.4% and 1.7% in the 4MB LLC setup, and the Prefetch-Hint scheme demonstrates the improvements by 13.5%, 12.7% and 0.5%**.** Like the within the shared-cache scheme unifying I/O data and CPU data, the Prefetch-Hint scheme is neither unable to reduce the cache interferences; moreover, it cannot benefit from the write-through policy. Thus the Prefetch-Hint scheme cannot outperform the shared-cache with WT scheme. In fact, Prefetch-Hint scheme exhibits the worst performance improvements in all schemes while DDC performs best and the speedups of DDC are about 3.4X of the Prefetch Hint.

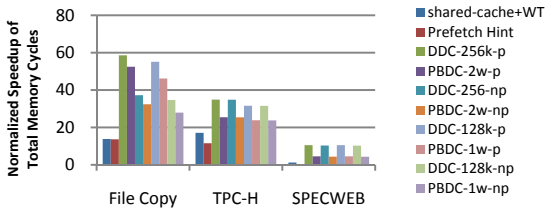

Figure 20 Normalized Speedup of Four Schemes (2MB LLC)

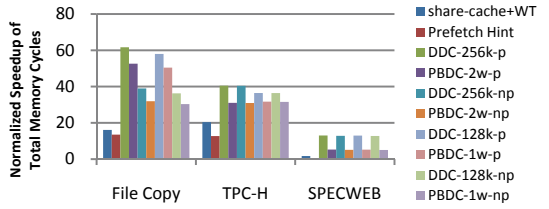

Figure 21 Normalized Speedup of Four Schemes (4MB LLC)

6. Related Work

Architectural approaches for improving I/O performance: there are numerous studies on caching I/O data. The concept of the cache injection means injecting I/O data into a CPU's cache directly [19, 21-23]. Because the injected I/O data and the CPU data share one cache, the approach is also called *shared-cache approach* in this paper. In the past several years, a few effective caching schemes have been proposed, e.g., Iyer proposing a chipset cache [15], Huggahalli et al. proposing the Direct Cache Access (DCA) scheme [14] and so on. Kumar et al. have shown the Prefetch-Hint scheme, which is a low complexity implementation of the DCA, can improve receive-side TCP/IP processing by 15.6%~43.4% [18, 17]. The DCA technology has been implemented in the Intel 82599 10 Gb Ethernet controller [4]. More recently, Berg has evaluated various I/O data coherence schemes in the embedded multicore systems [11]. However, as shown in Leon et al.'s study [19], shared-cache approach can even decrease application's performance. It should be noted that all these previous approaches unify I/O data and CPU data in one cache. In this paper, by analyzing the characteristics of I/O data, we propose architecturally separating I/O data from CPU data and present two concrete designs that perform better than previously proposed approaches.

Partitioned cache approaches: The partitioned cache approaches are proposed to reduce cache interferences among the threads in multicore platform. Previous studies mainly focus on what information to be used to partition a shared cache and how to partition the cache. Stone et al. [28] used information of change in cache misses to statically partition a cache for multiple applications. Suh et al. [30, 29] proposed dynamic partitioning of a shared cache based on position information of cache hit operations. Qureshi et al. [26] proposed utility-based cache partitioning (UCP), a low-overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses. Lin et al. [20] proposed an OS page coloring approach to partition a shared cache for multiple concurrent applications in multicore platform. Chiou [12] proposed a column-based cache partitioning scheme to enable some columns to act as scratchpad memories and to improve the predictability in a multitasking environment. Naz et al. [24] proposed a split cache approach for stream and scalar data and showed that it can significantly reduce miss rate. Ranganathan et al. [27] proposed a reconfigurable caches design that can dynamically divide a cache into multiple partitions for different usages such as TLB buffer, prefetched data and user-controlled memory. Albonesi [9] also proposed a minor-modification design for dynamical cache ways allocation. Overall, these studies indicate the low design cost of cache partition approaches. Nevertheless, there is little work on using the cache partitioning technique for improving I/O performance. We propose the PBDC scheme that applies the column-based cache partition technique to the processor LLC. Experimental results shown that this technique can effectively eliminate cache interferences between CPU data and I/O data and significantly improve I/O performance.

7. Conclusions

In this paper, we have proposed a DMA cache technique to separate I/O data and CPU data based on the observations of the different characteristics of the DMA and CPU memory reference behaviors. Concretely, the I/O data's produce-consume reuse distances inspire us to separate I/O data from CPU data and to consequently propose the DMA cache. The average sizes of various types of DMA requests indicate the choices of the DMA

cache sizes. The percentages of the sequential DMA memory references are used for the adoption of WT policy and the prefetch scheme for the DMA cache.

We have presented two concrete DMA cache designs, i.e., Decoupled DMA Cache (DDC) and Partition-Based DMA Cache (PBDC), which are for I/O-specific processors and general purpose processors respectively. By using an FPGA-based emulation platform, we have implemented and evaluated our designs and previous unified approaches. Experimental results show that both DDC and PBDC perform better than the existing approaches that use unified, shared caches for I/O data and CPU data.

Acknowledgment

We would like to thank Anand Sivasubramanium, Li Shang and the anonymous reviewers for their insightful feedback. We thank Kai Li, Yuanyuan Zhou, Lixin Zhang, Weisong Shi, Jiang Lin, Paolo Ienne, Yunji Chen and the ASL Group members for the valuable comments. This research is supported by the National Natural Science Foundation of China under grant numbers 60633040, 60925009, 60921002, 60903046, 60736012, 60673146 and 60603049, the National High Technology Research and Development 863 Program of China under grant numbers 2008AA110901, 2007AA01Z114 and 2007AA01Z112, the National Basic Research 973 Program of China under grant number 2005CB321600 and Beijing Natural Science Foundation under grant number 4072024.

Reference

[1] http://www.fusionio.com/PDFs/Pressrelease_Pressrelease_HP_1millionIOPS.pdf, 2009.
[2] http://www.intel.com/design/flash/nand/extreme/index.htm.
[3] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. 2007.
[4] *Intel® 82599 10 GbE Controller Datasheet*. July 2009.
[5] *The LEADING Play In On-Demand Network Services.* Mellanox.
[6] *SPECWEB2005.* http://www.spec.org/web2005.
[7] *The TPC benchmark H (TPC-H).* http://www.tpc.org/tpch/.
[8] *Xtreme System.* http://www.cadence.com/products/fv/xtreme series/Pages/default.aspx.
[9] David Albonesi. *Selective Cache Ways: On-Demand Cache Resource Allocation*. in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 1999.
[10] Yungang Bao, Mingyu Chen, Yuan Ruan, et al. *HMTT: a platform independent full-system memory trace monitoring system*. in *ACM SIGMETRICS,* 2008.
[11] T.B. Berg, *Maintaining I/O Data Coherence in Embedded Multicore Systems.* Micro, IEEE, 2009. **29**(3): p. 10-19.
[12] Derek Chiou, Prabhat Jain, Larry Rudolph, et al. *Application-specific memory management for embedded systems using software-controlled caches*. in *Proceedings of the 37th conference on Design automation*. 2000.
[13] Weiwu Hu, Jian Wang, Xiang Gao, et al., *Godson-3: A Scalable Multicore RISC Processor with x86 Emulation.* IEEE Micro, 2009. **29**(2): p. 17-29.
[14] Ram Huggahalli, Ravi Iyer, and Scott Tetrick, *Direct Cache Access for High Bandwidth Network I/O*, in *the Annual International Symposium on Computer Architecture*. 2005.
[15] R. Iyer. *Performance implications of chipset caches in web servers*. in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. 2003.
[16] David Patterson John Hennessy, *Computer Architecture: A Quantitative Approach， 3rd edition*. 2003: Morgan Kaufmann Publishers.
[17] A. Kumar, R. Huggahalli, and S. Makineni. *Characterization of Direct Cache Access on multi-core systems and 10GbE*. in *IEEE 15th International Symposium on High Performance Computer Architecture*. 2009.
[18] Amit Kumar and Ram Huggahalli. *Impact of Cache Coherence Protocols on the Processing of Network Traffic*. in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 2007.
[19] Edgar A. Leon, Kurt B. Ferreira, and Arthur B. Maccabe. *Reducing the Impact of the MemoryWall for I/O Using Cache Injection*. in *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*. 2007.
[20] Jiang Lin, Qingda Lu, Xiaoning Ding, et al. *Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems*. in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*. 2008.
[21] A. Milenkovic and V. Milutinovic. *Cache Injection on Bus Based Multiprocessors*. in *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*. 1998.
[22] Aleksandar Milenkovic and Veljko M. Milutinovic. *Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs*. in *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. 2000.
[23] V. Milutinovic, A. Milenkovic, and G. Sheaffer. *The cache injection/cofetch architecture: initial performance evaluation*. in *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 1997.
[24] Afrin Naz, Mehran Rezaei, Krishna Kavi, et al., *Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers.* ACM SIGARCH Computer Architecture News, 2005.
[25] Guru Venkataramani. Ioannis Doudalis. Yan Solihin. Milos Prvulovic. *FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation*. in *International Symposium on High-Performance Computer Architecture (HPCA-14),*. 2008.
[26] Moinuddin K. Qureshi and Yale N. Patt. *Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches*. in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 2006.
[27] Parthasarathy Ranganathan, Sarita Adve, and Norman Jouppi. *Reconfigurable Caches and their Application to Media Processing*. in *ISCA*. 2000.
[28] Harold S. Stone, John Turek, and Joel L. Wolf, *Optimal Partitioning of Cache Memory.* IEEE Trans. Comput., 1992. **41**(9): p. 1054-1068.
[29] G. E. Suh, L. Rudolph, and S. Devadas, *Dynamic Partitioning of Shared Cache Memory.* J. Supercomput., 2004. **28**(1): p. 7-26.
[30] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. *A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning*. in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. 2002.
[31] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen, *Architectural Exploiting Producer-Consumer Relationship in DMA Data Transfer to Improve I/O Performance.* Tech_Report, 2009.
[32] Steven P. Vanderwiel and David J. Lilja, *Data prefetch mechanisms.* ACM Comput. Surv., 2000. **32**(2): p. 174-199.
[33] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. *A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality*. in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 2000.