WARP: Memory Subsystem Effective for Wrapping Bursts of a Cache

Wooyoung Jang

State-of-the-art processors require increasingly complicated memory services for high performance and low power consumption. In particular, they request transfers within a burst in a wrap-around order to minimize the miss penalty of a cache. However, synchronous dynamic random access memories (SDRAMs) do not always generate transfers in the wrap-round order required by the processors. Thus, a memory subsystem rearranges the SDRAM transfers in the wrap-around order, but the rearrangement process may increase memory latency and waste the bandwidth of on-chip interconnects. In this paper, we present a memory subsystem that is effective for the wrapping bursts of a cache. The proposed memory subsystem makes SDRAMs generate transfers in an intermediate order, where the transfers are rearranged in the wrap-around order with minimal penalties. Then, the transfers are delivered with priority, depending on the program locality in space. Experimental results showed that the proposed memory subsystem minimizes the memory performance loss resulting from wrapping bursts and, thus, improves program execution time.

Keywords: Cache, SDRAM, Wrapping burst, System-on-chip.

I. Introduction

The latest processors require a memory subsystem to access synchronous dynamic random access memories (SDRAMs) in an increasingly complex manner as they execute a variety of heavy applications [1]. On the contrary, SDRAMs are showing significantly decreased efficiency owing to their inflexible operations and timing constraints as they operate at a high clock frequency to feed a number of processors [2]. Therefore, a memory system that can support various applications effectively and enhance memory efficiency has recently attracted much attention.

When a cache miss in the last-level cache occurs, a processor requests multiple words from a memory subsystem to replace a single cache block completely. It commonly uses burst-based transactions for receiving the words. That is, since a burst is composed of multiple transfers, it starts the first transfer at a given address and then continues throughout its length. The latest processors use an original program counter (PC) as the starting memory address via a memory management unit and requires transfers in a wrap-around order to reduce the miss penalty of a cache [3]. On the contrary, SDRAMs always generate transfers in a programmed order that depends on the burst length (BL), the burst type (BT), and the starting column address (CA). The programmed transfer order is different from the wrap-around transfer order required by a processor in the case where the CA is neither aligned nor half-aligned with a block of SDRAM columns. Table 1 shows the distribution of the three low-order bits of the column address CA[2:0] when the Princeton Application Repository for a Shared-Memory Computers (PARSEC) benchmark suite [4] was executed by MARSS-x86 [5]. MARSS-x86 requires, on average, a 79.2% aligned wrapping burst and a 2.4% half-aligned wrapping burst, neither of which causes any system performance loss. On the contrary, it requires, on average, an 18.4% unaligned wrapping

Manuscript received Oct. 2, 2016; revised Feb. 5, 2017; accepted Feb. 14, 2017. This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2014R1A1A1002262).

Wooyoung Jang (corresponding author, wyjang@dankook.ac.kr) is with the Department of Electronics and Electrical Engineering, Dankook University, Yongin, Rep. of Korea.

Table 1. Distribution of CA[2:0] in the PARSEC benchmarks.

Benchmark	CA[2:0] (%)							
	0	1	2	3	4	5	6	7
① x264	94.85	1.00	0.87	0.63	0.86	0.51	0.79	0.50
② Streamcluster	84.55	5.39	8.62	0.97	0.37	0.30	0.50	0.40
③ Fluidanimate	49.02	3.41	34.31	10.15	0.96	1.24	0.41	0.51
④ Canneal	43.64	7.09	13.74	0.78	13.04	0.75	14.89	6.08
⑤ Dedup	97.93	0.19	0.46	0.18	0.50	0.11	0.55	0.09
⑥ Vips	91.23	1.28	1.45	1.09	1.47	1.00	1.33	1.16
⑦ Bodytrack	85.92	2.21	2.74	1.64	2.16	1.56	1.88	1.89
⑧ Ferret	70.20	0.66	1.42	25.91	0.61	0.31	0.48	0.41
⑨ Freqmine	81.60	4.20	3.75	1.06	1.49	0.99	1.03	5.86
⑩ Swaptions	79.61	4.32	3.50	2.31	3.21	2.03	2.98	2.05
⑪ Facesim	82.47	4.58	3.90	1.64	2.08	1.45	1.99	1.90
⑫ Blackscholes	89.88	2.95	2.24	0.84	1.59	0.68	1.11	0.71
Average	79.24	3.10	6.42	3.93	2.36	0.89	2.29	1.77

burst, which can seriously degrade system performance. Thus, transfers within the unaligned wrapping burst must be rearranged by a memory subsystem before they are delivered to a processor. However, transfer rearrangement operations can gravely affect memory latency and efficiency.

In this paper, we present a memory subsystem that is effective for the wrapping bursts of a cache. In particular, the CA is not aligned with a block of SDRAM columns. The proposed memory subsystem modifies the starting memory address given by a processor, thus forcing the SDRAMs to generate transfers in an intermediate order. Transfers in the intermediate order are rearranged into the wrap-around order required by a processor with minimal penalty. Next, on the basis of the program locality in space, the proposed memory subsystem sends the transfers to the processors. Transfers including both a cache-missed word and words in an increasing address order from the cache-missed word should be delivered to shorten the memory latency, whereas transfers including words in a decreasing address order from the cache-missed word should be delivered to achieve a higher memory throughput. Since the proposed memory subsystem accesses SDRAMs in such a burst-aware manner and serves a processor in such a cache-aware manner, therefore, it can greatly improve the performance of the memories and, thus, the program execution time. Based on these concepts, the major novelty and contribution of this paper include the following.

- We propose a memory subsystem that is effective for the wrapping bursts of a cache. The CAs of the bursts are not aligned with a block of SDRAM columns.
- We analyze how many CAs of wrapping bursts are not aligned to a block of SDRAM columns in the PARSEC benchmark suite.
- We show the experimental results where the proposed memory subsystem, which is implemented using a cycle-accurate memory system simulator and a full-system x86 simulator, called DRAMSim2 [6] and MARSS-x86, respectively, considerably enhanced the memory latency and, thus, the program execution time compared with two conventional memory subsystems.

To the best of our knowledge, this is the first work that addresses the wrapping bursts of a cache in the case where their CAs are not aligned to a block of SDRAM columns. The rest of this paper is organized as follows. In the next section, we survey the related works. In Section III, we review a program execution on a simplified system-on-chip (SoC). In Section IV, we show the problem of conventional memory subsystems that do not consider a wrapping burst. In Section V, we present the proposed memory subsystem, which provides the wrapping bursts for a processor with minimal penalty. Section VI shows the experimental results. Finally, Section VII concludes this paper.

II. Previous Works

The latest applications require a processor to request for complicated memory services that can provide higher memory throughput and shorter memory latency. Thus, memory subsystems that can support such applications have become a key issue in SoC designs. To improve memory latency and throughput, the latest research studies have proposed various types of dynamic random access memory (DRAM) with a new architecture and organization. Asymmetric DRAM bank organizations for reducing the average main-memory access latency were proposed in [7]. An adaptive-latency DRAM with a simple mechanism for dynamically tailoring the DRAM timing parameters for the current operating condition was proposed in [8]. A tiered-latency DRAM with short bitlines was introduced in [9]. A half-DRAM proposed in [10] allows activating only half of a row to optimize the activation power consumption and bandwidth. A subarray-level parallelism architecture for reducing access latency from a different subarray in the same bank was proposed in [11]. In [12], a cache-inspired DRAM resilience architecture that substantially reduces the area and latency overheads of correcting bit errors in random locations due to faulty cells was proposed. In [13], a flexible DRAM that provides not only successive row access, but also successive column access, was proposed.

In addition, there have been many research studies on a memory subsystem for improving the system performance. A hardware mechanism called cache-conscious wavefront scheduling, which uses an intra-wavefront locality detector to capture a locality, was proposed in [14]. A new memory-buffer

chip called Centaur, which provides up to 128 MB of embedded DRAM buffer cache per processor along with an improved DRAM scheduler, was proposed in [15]. To minimize dependent cache miss latency, Hashemi and others [16] proposed adding enough functionality to dynamically identify instructions at the core and migrate them to the memory controller for execution. In [17], a dynamic scheduling algorithm was proposed for a set of sporadic real-time tasks that efficiently co-schedule a processor and a DMA execution to hide memory transfer latency. The design of a memory access scheduler for the type of memory traffic typically encountered with smart televisions was proposed in [18]. To improve the overall execution time, Ebrahimi and others [19] proposed a memory controller that manages inter-thread memory interference in parallel applications. In [20], the authors proved that cache partitioning leads to a lower hit rate and then proposed a cache-matching algorithm that captures data locality both within threads and across multiple threads.

These previous works do not consider the transfer order within a burst required by a processor. This is because they schedule memory requests and control the SDRAMs by a burst, but not a transfer. However, when a CA is not aligned to a block of SDRAM columns, SDRAM transfers are delivered in a programmed order that is different from the order required by a processor. Thus, the transfers should be rearranged before delivery, but the rearrangement operations may result in a critical system performance loss. On the contrary, the proposed memory subsystem hardly worsens memory latency and can shorten program execution time even when it provides wrapping bursts to a processor. This is because our work minimizes the penalty of rearrangement operations and serves transfers with priority, depending on the program locality in space.

III. Preliminaries

Figure 1 shows a simple example of a SoC architecture. The processors have a 64-bit instruction set architecture, and their last-level caches consist of 1 K 64-byte cache blocks. The processors are interconnected with memory subsystems via an on-chip interconnect with a 64-bit data bus. Suppose processor 1 executes the instructions of program 1. A PC starts at 64′h8000000000, which is translated to a physical memory address in the memory management unit. The first instruction whose physical memory address is 40′h8000000000 causes a cache miss. Processor 1 requests from memory subsystem 1 eight words starting at 40′h8000000000 in a wrap-around order. A wrap-around order is similar to an incrementing order in that the address for each transfer is an increment of the previous transfer address. However, in a wrap-around order, the address

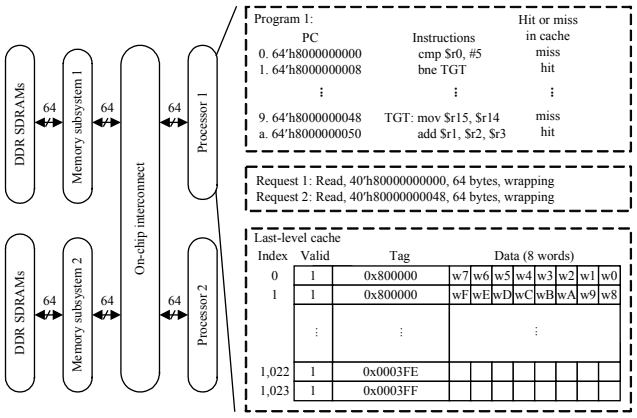

Fig. 1. Simple SoC architecture and program execution.

wraps around to a lower address when a wrap boundary is reached. The wrap boundary is the size of each transfer multiplied by the total number of transfers in a burst. Such a wrap-around order is useful for filling a cache block. After the memory latency period, the cache-missed word and seven words in an increasing address order from the cache-missed word are served to processor 1, and they are stored at w0 to w7 of the first cache block. Then, processor 1 can resume executing the instructions of program 1.

The second instruction, in which the PC is at 64′h800000008, is hit in the cache since it already exists in the second word of the first cache block (which is w1 in Fig. 1). The instruction is decoded as a branch, and we assume that the branch is taken. Thus, the next instruction is located at 64′h800000048, and it causes a cache miss. Processor 1 requests from memory subsystem 1 eight words starting at 40′h8000000048 in a wrap-around order. After the memory latency period, the words are served to processor 1, and they are stored at w9 to wF and then at w8 of the second cache block. Then, processor 1 can resume executing the instructions of program 1.

IV. Conventional Memory Subsystem: Wrapping Bursts Unaware

When referring to the information of memory request r_i, a memory subsystem should access the SDRAMs and then deliver the transfers ($t_{ij} \in WB_i$) to a processor, where WB_i represents a wrapping burst and j is a sequential number appended to each transfer. For example, the sequential number of the first, second, and last transfer generated by the SDRAMs is 0, 1, and BL-1, respectively. The three low-order bits of a given 40-bit memory address are considered as a byte offset since memory subsystem 1 is interconnected to the SDRAMs via an 8-byte data bus. The next 10 low-order bits of the memory address are used as a starting column address (CA_i). Table 2 shows the order of transfers ($O(CA_i[2:0],j)$) programmed

https://doi.org/10.4218/etrij.17.0116.0710

Table 2. Order of transfers within a burst in the case of BL8 and sequential BT [2].

Command	Starting CA[2:0]	Order of transfers within a burst (O(CA[2:0],j))							
		$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
Read	3′h0	0	1	2	3	4	5	6	7
	3′h1	1	2	3	0	5	6	7	4
	3′h2	2	3	0	1	6	7	4	5
	3′h3	3	0	1	2	7	4	5	6
	3′h4	4	5	6	7	0	1	2	3
	3′h5	5	6	7	4	1	2	3	0
	3′h6	6	7	4	5	2	3	0	1
	3′h7	7	4	5	6	3	0	1	2
Write	Don't care	0	1	2	3	4	5	6	7

in the SDRAMs [2] in the case where the BL and the BT of the SDRAMs are set to eight and a sequential type, respectively. Whereas write transfers should be delivered to the SDRAMs in a fixed order, read transfers are generated by the SDRAMs in a variable order, depending on CA_i[2:0]. Thus, such read transfers need to be rearranged in the wrap-around order that a processor requires. The rest of the given memory address is used as bank, row, rank, and channel addresses.

Memory subsystem 1, after receiving two memory requests, r_0 and r_1, from processor 1, starts accessing the SDRAMs at a given memory address. First, a row access strobe (RAS) command is issued to the SDRAMs for the activation of a single row selected. After activate to internal read/write delay time (t_{RCD}), a column access strobe (CAS) read command with CA_0 is issued to the SDRAMs. Figure 2 shows the operations of a conventional memory subsystem according to CA_i[2:0]. In the case where the starting address of memory request r_0 is 40′h8000000000, CA_0[2:0] is 3′h0. Thus, SDRAMs generate eight transfers in the order 0, 1, 2, 3, 4, 5, 6, and 7. Transfers 0, 1, 2, 3, 4, 5, 6, and 8 each include an 8-byte word stored at 40′h8000000000, 40′h8000000008, 40′h8000000010, 40′h8000000018, 40′h8000000020, 40′h8000000028, 40′h8000000030, and 40′h8000000038, respectively. Lastly, the transfers can be delivered to processor 1 in the order of arrival at a memory subsystem without any stop, as shown in Fig. 2(a). This is because a transfer order programmed in the SDRAMs is the same as the transfer order required by processor 1.

On the contrary, the SDRAMs do not generate transfers in the order that processor 1 requires for memory request r_1. The memory address of memory request r_1 is 40′h8000000048, and, thus, CA_1[2:0] is 3′h1. In the case where CA_i[2:0] is not 3′h0, a wrapping burst is not aligned to a block of SDRAMs. In the case where CA_1[2:0] is 3′h1, SDRAMs generate

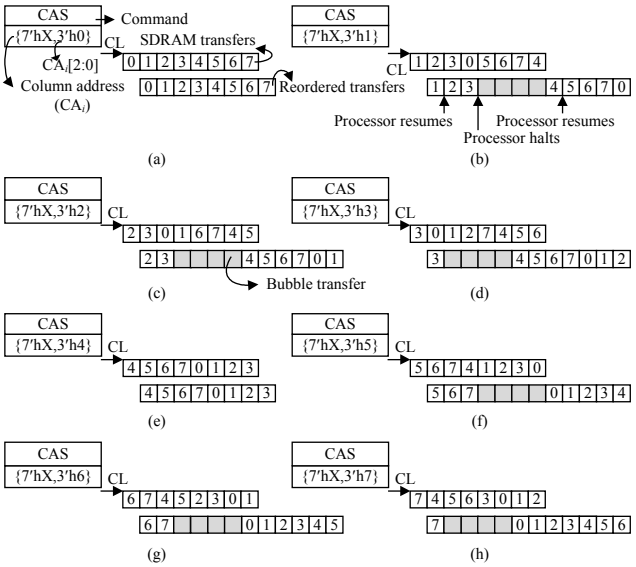

Fig. 2. Conventional memory subsystem supporting wrapping bursts: (a) CA_i[2:0] = 3′h0, (b) CA_i[2:0] = 3′h1, (c) CA_i[2:0] = 3′h2, (d) CA_i[2:0] = 3′h3, (e) CA_i[2:0] = 3′h4, (f) CA_i[2:0] = 3′h5, (g) CA_i[2:0] = 3′h6, and (h) CA_i[2:0] = 3′h7.

transfers in the order 1, 2, 3, 0, 5, 6, 7, and 4, as shown in Table 2. Transfers 1, 2, 3, 0, 5, 6, 7, and 4 each include a single word stored at 40′h8000000048, 40′h8000000050, 40′h8000000058, 40′h8000000040, 40′h8000000068, 40′h8000000070, 40′h8000000078, and 40′h8000000060, respectively. The transfers need to be rearranged before they are delivered to processor 1. The reason is that processor 1 awaits them in the order 1, 2, 3, 4, 5, 6, 7, and 0. As most instructions have spatial locality except when a branch is taken, the instructions in transfers 1 to 7 have a high possibility of being executed soon. The latest processors resume executing instructions for high performance even if all words in a cache block are not yet ready [3]. However, processor 1 halts again after performing the instructions in transfers 1 to 3. The reason is that transfers 4 to 7 are delayed for four cycles, as shown in Fig. 2(b). Therefore, it is effective to reduce the latency of transfers 1 to 7, but not that of transfers 1 to 3 within a burst in the case where CA_1[2:0] is 3′h1. If the on-chip interconnects adopt a winner-take-all bandwidth allocation that allocates all bandwidths to just one burst until it is finished or blocked before serving other bursts, then, their bandwidth can be seriously wasted. This is because so-called bubble transfers are delivered between transfer 3 and transfer 4. Therefore, transfers within a burst should be delivered without any break.

Figures 2(c) to (h) show the operations of a conventional memory subsystem when CA_i[2:0] is 3′h2 to 3′h7, respectively. A memory request where CA_i[2:0] is 3′h4 does not result in any performance penalty, as shown in Fig. 2(e). However, a memory request where CA_i[2:0] is 3′h2 or 3′h3 results in

longer program execution time and bandwidth loss of the on-chip interconnects, as shown in Figs. 2(b) and (c), respectively. In addition, a memory request where $CA_i[2:0]$ is 3′h5 to 3′h7 results in bandwidth loss of the on-chip interconnects, as shown in Figs. 2(f) to (h), respectively.

V. Proposed Memory Subsystem: Wrapping Bursts Aware

When a cache miss occurs, a cache-missed word should be rapidly provided to a processor. Moreover, words in an increasing address order from the cache-missed word within a cache block have a high possibility of being accessed soon, whereas words in a decreasing address order from the cache-missed word within a cache block have a low possibility of being accessed soon. The reason is that instructions within a program are sequentially executed except when a branch is taken. Therefore, transfers including not only a cache-missed word, but also words in an increasing address order from the cache-missed word within a cache block, should be delivered to shorten the memory latency. On the contrary, transfers including words in a decreasing address order from the cache-missed word should be delivered to achieve a higher memory throughput. With such observations, the proposed memory subsystem provides a wrapping burst for a processor with minimal performance penalty.

After receiving memory requests, the proposed memory subsystem starts accessing the SDRAMs. Like in the conventional memory subsystem, the proposed memory subsystem issues a RAS command to the SDRAMs for the activation of a single row selected. After t_{RCD}, it issues a CAS read command. However, the proposed memory subsystem does not use a given CA_i. Algorithm 1 shows how a given CA_i is modified. First, CA_i is copied into CA'_i. Then, $CA'_i[2:0]$ will be reset to 3′h0 if $CA_i[2:0]$ is less than 3′h4. Otherwise, $CA'_i[2:0]$ is not modified. The proposed memory subsystem sends CA'_i to the SDRAMs, as shown in Fig. 3. Thus, if $CA_i[2:0]$ is less than 3′h4, the proposed memory subsystem will receive transfers from the SDRAMs in the order 0, 1, 2, 3, 4, 5, 6, and 7. Otherwise, it will receive transfers in a variable order programmed in the SDRAMs.

Algorithm 1. Starting column address modification
Input: starting column address (CA_i) of memory request r_i
Output: modified starting column address (CA'_i)

CA_i is copied into CA'_i;
if three low-order bits of CA_i ($CA_i[2:0]$) are less than four **then**
    Only $CA'_i[2:0]$ is reset to 3′h0;
end if
SDRAMs are accessed with CA'_i as shown in Fig. 3;

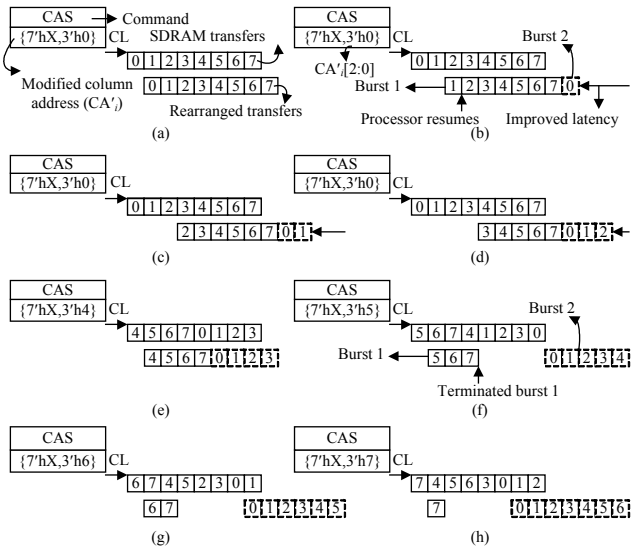

Fig. 3. Proposed memory subsystem supporting wrapping bursts: (a) $CA_i[2:0]$ = 3′h0, (b) $CA_i[2:0]$ = 3′h1, (c) $CA_i[2:0]$ = 3′h2, (d) $CA_i[2:0]$ = 3′h3, (e) $CA_i[2:0]$ = 3′h4, (f) $CA_i[2:0]$ = 3′h5, (g) $CA_i[2:0]$ = 3′h6, and (h) $CA_i[2:0]$ = 3′h7.

Figure 3 shows the operations of the proposed memory subsystem according to $CA_i[2:0]$. In the case where $CA_i[2:0]$ is less than 3′h4, the SDRAMs output transfers 0 to 7 in an increasing order, as shown in Figs. 3(a) to (d). For example, if the memory address of memory request r_1 is 40′h8000000048, $CA_1[2:0]$ is 3′h1. Since the proposed memory subsystem resets $CA_1[2:0]$ to 3′h0 via Algorithm 1, the SDRAMs generate transfers in the order 0, 1, 2, 3, 4, 5, 6, and 7, as shown in Fig. 3(b). Such an intermediate order can be rearranged to the order 1, 2, 3, 4, 5, 6, 7, and 0 as required by processor 1, with minimal performance penalty and design cost.

Algorithm 2 shows how our memory subsystem rearranges transfers in any wrap-around order required by a processor. First, $CA_i[2:0]$ is copied into *wrap_around_order* and j is reset to 0. Whenever any transfer $t_{ij} \in WB_i$ is delivered to our memory subsystem, the order of transfers ($O(CA_i[2:0], j)$) in Table 2 is referred. Then, a transfer can be included in burst 1 and then delivered to a processor if its $O(CA_i[2:0], j)$ is equal to *wrap_around_order*. Burst 1 is given its own priority for a fast service since it includes a cache-missed word and words in an increasing address order from the cache-missed word, which have a high program locality in space. Then, *wrap_around_order* will increase by 1 if it is not equal to *BL*-1. Otherwise, *wrap_around_order* is reset to 0 and burst 1 is terminated. On the contrary, a transfer will be stored in a buffer (B) if its $O(CA_i[2:0], j)$ is not equal to *wrap_around_order*. Since the transfer includes a word in a decreasing address order from the cache-missed word, it has a low possibility of being used soon. Then, j increases by 1. These procedures are repeated until

Algorithm 2. Transfer rearrangement

Input: $CA_i[2:0]$, $CA'_i[2:0]$, $t_{ij} \in WB_i$ in an order programmed in the SDRAMs, Table 2, *BL*

Output: $t_{ij} \in WB_i$ in the wrap-around order required by a processor

```
wrap_around_order ← CA_i[2:0];
j ← 0;
if t_ij from SDRAMs then
  if O(CA'_i[2:0], j) is equal to wrap_around_order then
    t_ij is included in burst 1 and delivered with its own priority;
    if wrap_around_order is equal to BL-1 then
      wrap_around_order ← 0;
      burst 1 is terminated
    else
      wrap_around_order ← wrap_around_order + 1;
    end if
  else
    t_ij is stored in a buffer (B);
  end if
  j ← j +1;
end if
for t_ij ∈ B do
  if O(CA_i[2:0], j) is equal to wrap_around_order then
    t_ij is included in burst 2 and delivered with low priority;
    wrap_around_order ← wrap_around_order + 1;
  end if
end for
```

all transfers for memory request r_i are received from the SDRAMs. Next, transfers in a buffer should be rearranged and then delivered. $O(CA_i[2:0], j)$ of all transfers in a buffer is compared with *wrap_around_order*. If $O(CA_i[2:0], j)$ of any transfer is equal to *wrap_around_order*, the transfer can be included in burst 2 and then delivered to a processor. Burst 2 is given a low priority for a best-effort service since it includes words in a decreasing address order from the cache-missed word. The words have a low possibility of being used soon. Then, *wrap_around_order* increases by 1. These procedures are repeated until there is no more transfer in the buffer.

For example, suppose SDRAM transfers are generated for memory request r_1, as shown in Fig. 3(b). $CA_1[2:0]$ is copied into *wrap_around_order*, and *j* is reset to 0. $O(CA_1[2:0], j)$ of the first transfer is 0 in Table 2 when $CA_1[2:0]$ is 3′h1 and *j* is 0. The first transfer is stored in a buffer since its $O(CA_1[2:0], j)$ is not equal to *wrap_around_order*. Next, $O(CA_1[2:0], j)$ of the second transfer is 1 in Table 2 when $CA_1[2:0]$ is 3′h1 and *j* is 1. The second transfer is included in burst 1 with its own priority and delivered to processor 1 since its $O(CA_1[2:0], j)$ is equal to *wrap_around_order*. Then, both *wrap_around_order* and *j* increase by 1. These procedures are repeated until all transfers for memory request r_1 are received from the SDRAMs. As a result, transfers 1 to 7 included in burst 1 with their own priority are delivered to processor 1, and only transfer 0 is stored in a buffer, as shown in Fig. 3(b). At that time, *wrap_around_order* is reset to 3′h0 and burst 1 is terminated. Next, all transfers in the buffer should be delivered to processor 1. Transfer 0 can be included in burst 2 with a low priority and then delivered to processor 1 since its $O(CA_1[2:0], j)$ is equal to *wrap_around_order*.

As shown in Figs. 2(b) and 3(b), the proposed memory subsystem delivers transfers 1 to 3 one cycle slower, but delivers other transfers three cycles faster than the conventional memory subsystem. Since the overall program execution time depends not only on the latency of transfers 1 to 3, but also on the latency of transfers 4 to 7, therefore, it is greatly improved by our memory subsystem. Moreover, transfers 1 to 7 including both a cache-missed word and words in an increasing address order from the cache-missed word are delivered without any break. Since the on-chip interconnects no longer deliver bubble transfers, as a result, their effective bandwidth can be enhanced.

When a given $CA_i[2:0]$ is greater than 3′h0 and less than 3′h4, the proposed memory subsystem causes a longer latency of transfers including a cache-missed word and some words in an increasing address order from the cache-missed word, but achieves a shorter latency of transfers including the others in an increasing address order from the cache-missed word compared to the conventional memory subsystem. However, the longer latency of some transfers is hidden behind the shorter latency of all transfers by our memory subsystem. The reason is that the overall program execution time depends on the latency of a missed word and all words in an increasing address order from the cache-missed word owing to the program locality in space, except when the words include a branch instruction that is taken. Finally, the proposed memory subsystem can improve memory latency by up to one to three cycles and save the bandwidth of the on-chip interconnects by up to four cycles.

When a given $CA_i[2:0]$ is equal to or greater than 3′h4, the proposed memory subsystem accesses the SDRAMs without the modification of CA_i, as shown in Figs. 3(e) to (h). For example, in the case where a given $CA_i[2:0]$ is 3′h5, it is not reset to 3′h0, as shown in Fig. 3(f). Thus, the SDRAMs generate transfers in the order 5, 6, 7, 4, 1, 2, 3, and 0, as shown in Table 2. Transfers 5 to 7 are included in burst 1 and delivered as their $O(CA_1[2:0], j)$ is equal to *wrap_around_order*. Since transfers 5 to 7 including a cache-missed word and words in an increasing address order from the cache-missed word have a high possibility of being used soon, burst 1 is delivered with its own priority. On the contrary, transfers 4, 1, 2, 3, and 0 are stored in a buffer. The transfers can be included in burst 2 and delivered when their $O(CA_1[2:0], j)$ is equal to

wrap_around_order. Since transfers 0 to 4 including words in a decreasing address order from the cache-missed word have a low possibility of being used soon, burst 2 is delivered with a low priority. The on-chip interconnects can deliver other urgent bursts between burst 1 and burst 2 instead of delivering bubble transfers. Burst 2 will be delivered when the on-chip interconnects are not busy with urgent bursts. Thus, the proposed memory subsystem can save the bandwidth of the on-chip interconnects by up to four cycles per memory request in the case where $CA_i[2:0]$ is 3′h5.

In the case where a given $CA_i[2:0]$ is equal to or greater than 3′h4, the proposed memory subsystem delivers both a missed word and words in an increasing address from the cache-missed word with no performance penalty. Since our memory subsystem assigns a low priority to transfers including words in a decreasing address order from the cache-missed word, therefore, other urgent transfers are less blocked by transfers with a low priority. Therefore, the proposed memory subsystem can greatly improve the overall program execution time, especially in multi-core processors performing a number of applications at the same time.

VI. Experimental Results

We used MARSS-x86 [5] and DRAMSim2 [6] to evaluate the proposed memory subsystem on unaligned wrapping bursts. DRAMSim2 and MARSS-x86 are a cycle-accurate memory system simulator for double data rate (DDR) 2/3 SDRAMs and a cycle-accurate full system simulator for multicore x86 CPUs, respectively. DRAMSim2 was configured for a DDR3-1600 SDRAM with one channel and two ranks per channel, as shown in Table 3. Then, DRAMSim2 was modified with Algorithms 1 and 2 to make it effective for unaligned wrapping bursts, called WARP. The proposed algorithms in the modified DRAMSim2 were applied to memory requests for instructions, whereas conventional algorithms were applied to memory requests for data. This is because data have weak spatial and temporal localities. The proposed algorithms can be applied to memory requests for prefetching even if the memory requests are neither urgent nor useful. This is because the proposed algorithms can save the bandwidth of the on-chip interconnects, thanks to the few bubble transfers. MARSS-x86 was configured for a quad-core processor that includes a 64 KB two-way set associative L1 cache and a 2 MB eight-way set associative L2 cache, as shown in Table 3. The program execution time and memory latency of WARP were compared with those of two conventional memory subsystems. One of the conventional memory subsystems [3], called CONV1, used an original $CA_i[2:0]$ and then rearranged the transfers in the order required by a processor, as shown in Fig. 2. The other

Table 3. Processor and memory configurations.

Processor	Quad-core, 3 GHz, out-of-order, 4 issues per core, 4 threads
L1 I/D-cache	64 kB, 2-way associativity, private, 64 B block size, 2-cycle latency, least recently used (LRU)
L2 cache	2 MB, 8-way associativity, shared, 64 B block size, 5-cycle latency, pseudo LRU
Memory subsystem	First-ready, first-come first-served (FR-FCFS), "row:bank:rank:channel:column" mapping, 64-entry queue, open page policy
SDRAM	1 channel, 2 ranks per channel, 8 GB, 64 b data width, DDR3-1600 (t_{RCD} = 13.75 ns, t_{RP} = 13.75 ns, t_{CL} = 13.75 ns)

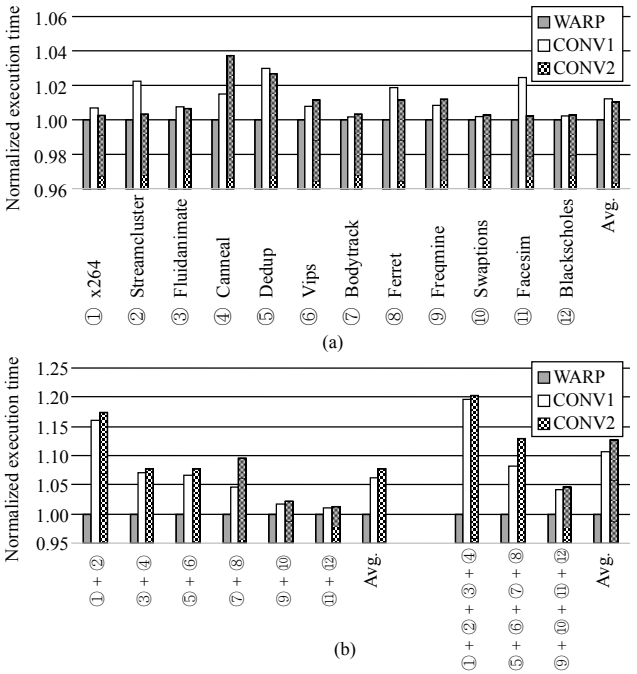

Fig. 4. Comparison of program execution time in the PARSEC benchmarks: (a) single benchmark running on the quad-core processor and (b) multiple benchmarks running simultaneously on the quad-core processor.

conventional memory subsystem [5], called CONV2, modified $CA_i[2:0]$ into 3′h0 and then rearranged the transfers in the order required by a processor. We conducted our evaluations by running the PARSEC benchmarks [4].

Figure 4 shows the comparison of program execution time normalized by our WARP. In Fig. 4(a), each benchmark was individually executed by MARSS-x86. WARP achieved 1.2% and 1.0% shorter program execution time than those of CONV1 and CONV2, respectively, on average. This result will be similar to that of a single-core processor. WARP showed a small improvement in program execution time in the case where a single benchmark was executed. The reason is that the ratio of memory requests to instructions in the PARSEC benchmarks was just 0.4% on average, thanks to the high cache-hit rate. In

https://doi.org/10.4218/etrij.17.0116.0710

such a non-intensive memory benchmark, memory requests have a low possibility of interfering with each other, and, thus, the performance loss resulting from unaligned wrapping bursts is neither propagated nor accumulated.

MARSS-x86 executing a fluidanimate benchmark required more than 50% unaligned wrapping bursts, but WARP achieved just 0.73% and 0.66% shorter program execution time than those of CONV1 and CONV2, respectively. On the contrary, MARSS-x86 executing a dedup benchmark required 1.53% unaligned wrapping bursts, but WARP achieved 3.00% and 2.68% shorter program execution time than those of CONV1 and CONV2, respectively. WARP showed more improved program execution time for a dedup benchmark than for a fluidanimate benchmark. This is because a dedup benchmark causes more memory requests than those of a fluidanimate benchmark with irregular intervals, even though a dedup benchmark has a lower ratio of unaligned wrapping bursts to all wrapping bursts than that of a fluidanimate benchmark. Therefore, we expect that the proposed WARP will achieve better improvement in program execution time in the case where memory-intensive benchmarks are executed.

Figure 4(b) shows the comparison of program execution time in the case where two or four benchmarks were executed by MARSS-x86 at the same time. A quad-core processor accesses SDRAMs more often and irregularly than a single-core processor executing one benchmark at a time. A number of memory requests by the quad-core processor have a high possibility of interfering with each other, and, thus, the performance loss resulting from unaligned wrapping bursts can be either propagated or accumulated. WARP achieved 6.2% and 7.7% shorter program execution time than those of CONV1 and CONV2, respectively, when two benchmarks were executed at the same time. In addition, it achieved 10.7% and 12.6% shorter program execution time than those of CONV1 and CONV2, respectively, when four benchmarks were executed at the same time. Thus, the proposed memory subsystem can be more effective for a multi-core processor executing multiple benchmarks at the same time than a single-core processor executing one benchmark at a time.

Figure 5 shows the comparison of memory latency normalized by WARP. Since memory latency was measured just for memory requests, the performance of the memory subsystems was evaluated fairly. In Fig. 5(a), WARP achieved 4.95% and 4.19% shorter memory latency than those of CONV1 and CONV2, respectively, when MARSS-x86 executed only a single benchmark. Figure 5(b) shows the comparison of memory latency when two or four benchmarks were executed by MARSS-x86 at the same time. Our WARP achieved 19.88% and 21.35% shorter memory latency than those of CONV1 and CONV2, respectively, when MARSS-

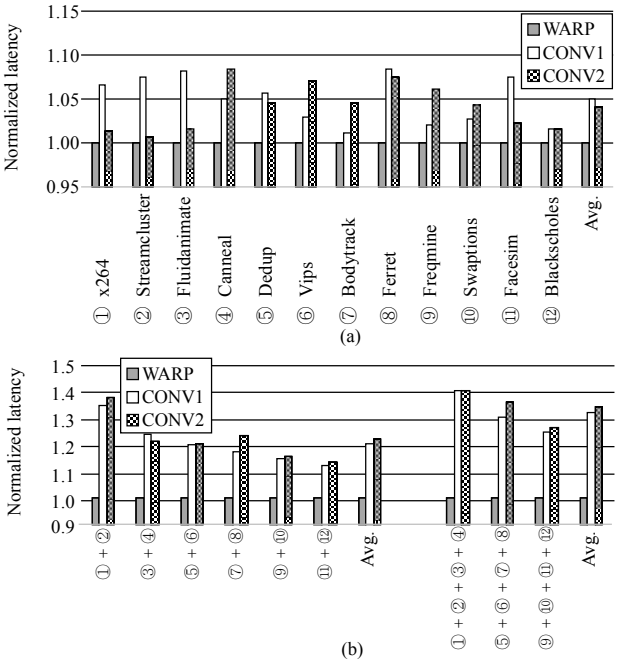

Fig. 5. Comparison of memory latency in the PARSEC benchmarks: (a) single benchmark running on the quad-core processor and (b) multiple benchmarks running simultaneously on the quad-core processor.

x86 executed two benchmarks at the same time. In addition, WARP achieved 31.14% and 33.34% shorter memory latency than those of CONV1 and CONV2, respectively, when MARSS-x86 executed four benchmarks at the same time. Thus, we expect that the proposed WARP will achieve better improvement in program execution time in the case where a number of benchmarks are executed at the same time.

The design cost of the proposed approach is similar to that of the conventional approach since our approach does not require any expensive storage of elements. Algorithm 1 can be implemented with simple combinational logic circuits since it resets CA[2:0] to 3′h0 depending on CA[2]. The delay of the combinational logic circuits can be ignored since it is not related to a critical pass. Algorithm 2 is similar to the transfer rearrangement algorithm of the conventional approach, but it has an additional function for generating burst 2. However, since the hardware for the additional function is used to configure the on-chip interconnect parameters for burst 2, it can also be implemented with combinational logic circuits. Whereas the latest memory subsystems are implemented with hundreds of thousands of gates, the combinational logic circuits for Algorithms 1 and 2 can be implemented with only tens of gates. Thus, the overhead for implementing our algorithms makes up only an extremely small part of the total memory subsystem. Since the on-chip interconnect parameters can be configured in advance, their hardware has little impact on the

static timing analysis.

VII. Conclusion

This paper presented a memory subsystem that is effective for wrapping bursts. Such wrapping bursts are mainly required for reducing the miss penalty of the last-level cache. Since SDRAMs generate transfers in a programmed order, memory subsystems should rearrange the transfers in the wrap-around order required by a processor. However, such operations can cause a critical performance loss. The proposed memory subsystem enables SDRAMs to generate transfers in an intermediate order, where the transfers are rearranged in a wrap-around order with minimal performance losses. Then, the transfers are delivered with a priority, depending on the program locality in space. Experimental results showed that our approach greatly improves the program execution time and memory latency. In conclusion, the proposed memory subsystem provides more opportunities to speed up the latest processors with high memory performance.

References

[1] ITRS, "International Technology Roadmap for Semiconductors," 2013.

[2] JEDEC, DDR 1, 2, 3, and 4 SDRAM Standard, Accessed 2016. http://www.jedec.org

[3] ARM, ARM Processor Architecture, Accessed 2015. http://www.arm.com

[4] C. Bienia et al., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Int. Conf. Parallel Archit. Compilation Techn.*, Toronto, Canada, Oct. 25–29, 2008, pp. 72–81.

[5] A. Patel et al., "MARSS: a Full System Simulator for Multicore x86 CPUs," *ACM/IEEE Des. Autom. Conf.*, San Diego, CA, USA, June 5–10, 2011, pp. 1050–1055.

[6] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: a Cycle Accurate Memory System Simulator," *Comput. Archit. Lett.*, vol. 10, no. 1, Jan. 2011, pp. 16–19.

[7] Y.H. Son et al., "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," *IEEE Int. Symp. Comput. Archit.*, Tel Aviv, Israel, June 23–27, 2013, pp. 380–391.

[8] D. Lee et al., "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," *IEEE Int. Symp. High Performance Comput. Archit.*, San Francisco, CA, USA, Feb. 7–11, 2015, pp. 489–501.

[9] D. Lee et al., "Tiered-Latency DRAM: a Low Latency and Low Cost DRAM Architecture," *IEEE Int. Symp. High Performance Comput. Archit.*, Shenzhen, China, Feb. 23–27, 2013, pp. 615–626.

[10] T. Zhang et al., "Half-DRAM: a High-Bandwidth and Low-Power DRAM Architecture form the Rethinking of Fine-Grained Activation," *IEEE/ACM Int. Symp. Comput. Archit.*, Minneapolis, MN, USA, June 14–18, 2014, pp. 349–360.

[11] Y. Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," *Annu. Int. Symp. Comput. Archit.*, Portland, OR, USA, June 9–13, 2012, pp. 368–379.

[12] Y.H. Son et al., "CiDRA: a Cache-Inspired DRAM Resilience Architecture," *IEEE Int. Symp. High Performance Comput. Archit.*, San Francisco, CA, USA, Feb. 7–11, 2015, pp. 502–513.

[13] J. Yu and W. Jang, "FDRAM: DRAM Architecture Flexible in Successive Row and Column Accesses," *IEEE Int. Conf. Comput. Des.*, New York, USA, Oct. 18–21, 2015, pp. 480–483.

[14] T.G. Rogers, M. O'Connor, and T.M. Aamodt, "Cache-Conscious Wavefront Scheduling," *IEEE/ACM Int. Symp. Microarchit.*, Cambridge, UK, Dec. 1–5, 2014, pp. 72–83.

[15] W.J. Starke et al., "The Cache and Memory Subsystems of the IBM POWER8 Processor," *IBM J. Res. Develop.*, vol. 59, no. 1, Jan./Feb. 2015, pp. 3:1–3:13.

[16] M. Hashemi et al., "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," *Annu. Int. Symp. Comput. Archit.*, Seoul, Rep. of Korea, June 18–22, 2016, pp. 444–455.

[17] S. Wasly and R. Pellizzoni. "Hiding Memory Latency Using Fixed Priority Scheduling," *IEEE Real-Time Embedded Technol. Applicat. Symp.*, Berlin, Germany, Apr. 15–17, 2014, pp. 75–86.

[18] C.H. Hahm et al., "Memory Access Scheduling for a Smart TV," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 26, no. 2, Feb. 2016, pp. 399–411.

[19] E. Ebrahimi et al., "Parallel Application Memory Scheduling," *IEEE/ACM Int. Symp. Microarchit.*, Porto Alegre, Brazil, Dec. 3–7, 2011, pp. 362–373.

[20] W. Zhang, F. Liu, and R. Fan, "Cache Matching: Thread Scheduling to Maximize Data Reuse," *Proc. High Performance Comput. Symp.*, Tampa, FL, USA, Apr. 13–16, 2014, pp. 47–54.

Wooyoung Jang received his BS degree in radio science and technology engineering from Kyunghee University, Yongin, Rep. of Korea, in 1998; his MS degree in electrical and computer engineering from Yonsei University, Seoul, Rep. of Korea, in 2000; and his PhD degree in electrical and computer engineering from the University of Texas at Austin, USA, in 2011. From 2000 to 2013, he was a senior engineer with the System Large Scale Integration Division, Samsung Electronics, Yongin, Rep. of Korea. He is currently an assistant professor with the Department of Electronics and Electrical Engineering, Dankook University, Yongin, Rep. of Korea. His research interests are computer architecture, embedded systems, and low power design.