CBM: Core Based Memory Scheduling method

Kohei Hosokawa[1] Yasuo Ishii[1,2] Mary Inaba[1] Kei Hiraki[1]

Abstract: In modern chip-multiprocessor systems, DRAM is shared among multiple threads. The memory scheduler must resolve the inter-thread contention for the DRAM effectiveness. Previously proposed DRAM memory schedulers have calculated the memory access intensity of each thread for the priority scheduling. Existing methods[2], [3] analyze the number of memory requests served in memory controller to get memory-intensity, but these methods lack prediction accuracy. TCM[1] avoids this problem by using MPKI information. TCM can improve the prediction accuracy, but takes very long cycles to update the thread priority. As a result, TCM lacks the timeliness of the priority prediction.
This paper presents a new memory scheduling method, Core-Based Memory scheduling(CBM), which utilizes core information for memory-intensity evaluation of each thread. Our key idea is 1) to refer the distance of instruction count between each memory request for the priority calculation, and 2) to place the priority scheduler on each core.
CBM judges the core calculation progress by comparing the instruction counter distance between the new memory request and the last one. By doing so, CBM can utilize the instruction progress information of each core directly, thus we can predict the memory-intensity more accurately.
CBM also proposes to calculate the thread priority not on the memory controller but on the private cache of each core. By doing so, even in concurrent many-channel memory system, CBM can decide priority without the heavy inter-channel communication. Therefore, CBM accomplishes high timeliness on the priority update.
We evaluate CBM by using the workloads of Memory Scheduling Championship (MSC) and compare its performance to two existing scheduling algorithms. We found that CBM achieves both the best throughput and fairness.

Keywords: Memory Scheduling, Prediction

1. Introduction

Off-chip DRAM memory has been one of the major bottleneck of processing due to the higher latency compared to CPU. This heavy latency gets even longer in chip-multiprocessor(CMP) systems, in which DRAM memory is shared among multiple threads. The inter-thread contention in CMP makes the DRAM performance worse: each thread issues memory requests simultaneously, which wasts spatial and temporal locality with each other and even causes starvation. To enhance the modern DRAM memory system efficiently, it is inevitable to handle this inter-thread contention.

There are three metrics mainly used to evaluate the effectiveness of memory scheduling algorithms: fairness, system throughput, and energy consumption. First, the fairness is important for the multi-thread workloads, otherwise the most delayed thread will slow down the overall system performance. Second, the energy consumption should be low because the DRAM energy consumption is not negligible on the modern CMP systems. Of course, the system throughput should be high for the effective DRAM usage.

To achieve the goal of high system throughput and high fairness, several scheduling algorithms have been proposed. State of the art schedulers exploit the memory-intensity of threads to improve system throughput. In these methods, memory scheduler takes the strategy of prioritizing the memory-non-intensive threads over the memory-intensive threads to reduce the total core stall time in overall system while keeping fairness high. However, when updating the thread priority, existing methods[1], [2], [3] need heavy inter-channel communication periodically to gather the statistical information distributing among every channel. This long time communication and bandwidth overhead restricts the priority update frequency low, resulting in the lack of priority prediction timeliness. Moreover, [2], [3] use the Served Requests Per Cycle(SRPC) on the memory scheduler to estimate the *memory-intensity per instruction* of each thread. This SRPC-based strategy cannot distinguish the core calculation time and stall time, so the estimated *memory-intensity per instruction* number is not accurate.

Our new scheduling algorithm exploits the core information of each thread to calculate the thread priority based on two key strategies. First, we use the instruction count distance of each memory request to calculate the memory-access-intensity. We can evaluate accurate memory-intensity by referring the instruction count because the instruction count describes the core progress accurately. Therefore, our scheduler can improve the accuracy of memory-intensity estimation. Second, we move the priority scheduling system from memory scheduler to private cache of each core. By placing priority scheduler on core side, the priority scheduler can detect all the memory requests from a core. This means that the priority scheduler does not need to communicate all memory channels to gather statistical information of

[1] The university of Tokyo
[2] NEC

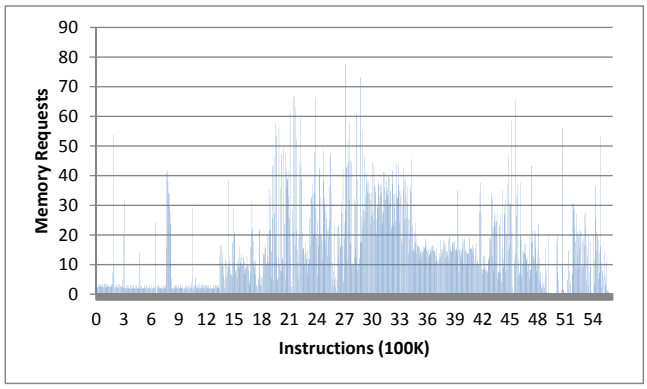


Fig. 1 The number of the LLC Miss per 100 Kilo Instruction on Blackscholes workload. The LLC miss rate changes as the calculation proceeds. We can see the clear tendency of memory-intensity in LLC Miss per Instruction statistics.

memory requests from a core. Without heavy inter-channel synchronization communication, CBM can update thread priority on every memory access happening, enabling more timely priority control.

In this paper, we make the following contributions: 1)We use the ”instruction counter distance” of each memory access for the calculation of recent memory-intensity. This method can capture the immediate change of the thread’s memory-intensity state between non-memory-intensive phase and memory-intensive phase. 2)We change the placement of priority scheduler from memory scheduler to private cache of each core. This avoids the heavy inter-channel communication of gathering statistical information for priority scheduling. Therefore, we can update the thread priority at the occurrence of each memory request without suffering inter-channel priority inconsistency.

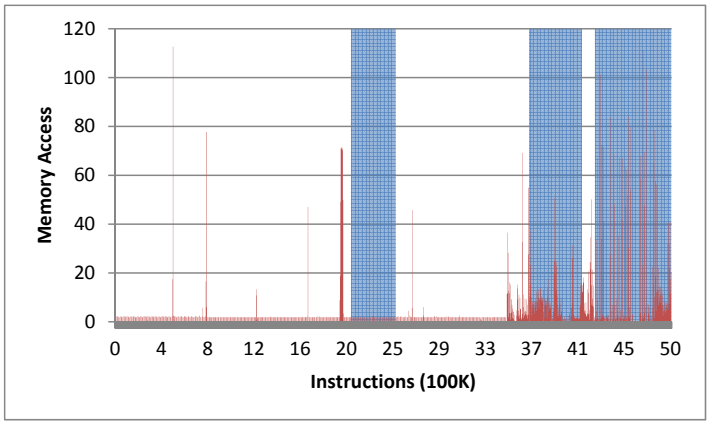
2. Background and Motivation

2.1 Memory Access statistics

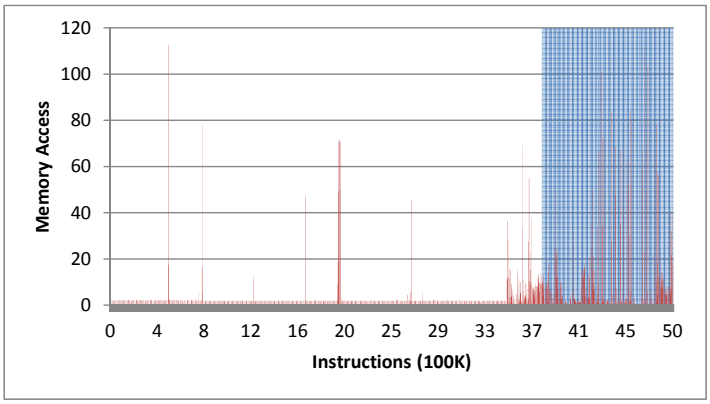
Previous paper has showed that system throughput improves by prioritizing low memory-intensity threads. Low memory-intensity thread spends most of the time in calculation, so the instruction throughput(Instruction Per Cycle) is high. In such case, by prioritizing its memory requests and serving them with low latency, the thread progresses far. On the other hand, memory-intensive thread progresses relatively smaller per a memory request and stalls immediately. In such case, the memory-intensive thread will progress by rather enhancing row-hit rate and memory access throughput than improving latency low. Therefore, it is important to prioritize non-memory-intensive thread for high system throughput.

Previous schedulers evaluated memory-intensity by analyzing the number of SRPC(served requests per cycle) or MPKI(miss per kilo instruction). The figure 1 shows the MPKI data in Blackscholes workload. In figure 1, we can see that the memory-intensity changes between heavy and light for the progress of instruction counter. By prioritizing a thread during the non-memory-intensive phase, the thread will progress far without stalling due to the long LLC miss.

This MPKI tendency is hard to be recognized by analyzing the number of SRPC on memory controller. The figure 2 shows the



(a) TCM (quanta length: 500K cycles)



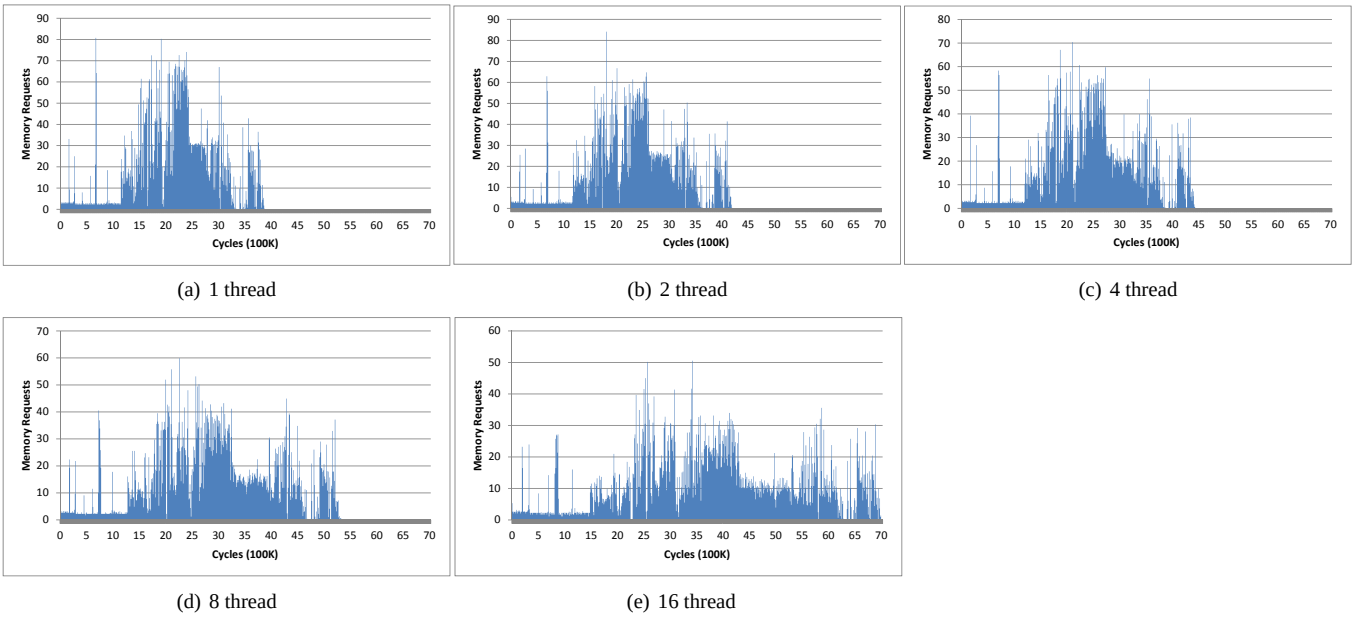
(b) TCM (quanta length: 1M cycles)

Fig. 4 Priority transition of TCM. The part with blue background is judged as the memory-intensive phase, and the part with white background as the non-memory-intensive phase. TCM scheduling lacks timeliness due to the coarse update policy, so the phase recognition in figure(a) is wrong near 2.2M instructions and 3.6M instructions. However, with coarser priority update policy in figure(b), it also lacks accuracy.

relationship between SRPC and the number of threads running at a time. SRPC result gets spread uniformly as the thread number increases, and memory-intensity tendency gets hard to be recognized. Moreover, in figure 2(e), there happens a blank-request time caused by the core stall time due to the inter-thread interference. This blank time is to be judged as memory-intensive phase, but the judgment is difficult in the multi-channel memory scheduler (We will show more details in the next subsection). As a result, memory-intensity tendency is unrecognizable by only using SRPC information and not using the core stall information or MPKI.

2.2 Inter-channel communication and timeliness

Modern DRAM memory has multiple memory channels separated from each other, so the memory requests which arrive at a memory channel are invisible to the other memory channels. [2] showed that inter-channel synchronization between memory channels is important for the high system throughput. For example, even if a channel A serves thread P eagerly, but if channel B does not prioritize thread P, then the final progression of thread P is delayed by channel B. In such situation, the system throughput will get worse due to the delayed threads besides thread P in channel A. Taken together, prioritizing a thread without synchroniz-



(a) 1 thread (b) 2 thread (c) 4 thread

(d) 8 thread (e) 16 thread

Fig. 2 Served Requests per Cycle(SRPC) statistics of the instructions shown in figure 1. We used Close Page Policy scheduler as the memory scheduler. This graph is derived from one of four memory channels. Different from figure 1, the memory-intensity tendency gets unclear on SRPC statistics as the number of running threads increases. Moreover, we can see the blank period near 2,300K Cycles in figure 2(e). This blank is caused by the inter-thread interference, and should be recognized as *memory-intensive*. However, memory scheduler cannot judge whether this period is caused by inter-thread interference or not without heavy inter-channel communication.

ing with other channels worsens system throughput. Moreover, if each memory controller analyses only the information available in its own channel, it leads to severe mis-prediction as is shown in figure 6. The inter-channel bias of the request arrival number is not negligible and causes wrong memory-intensity judgment. Therefore, it hurts performance to update priority without synchronization process.

For this reason, [1], [3] estimate the MPKI or SRPC by gathering into meta scheduler the memory access history distributing among memory channels and LLC. This gathering operation needs heavy communication cost(both latency and bandwidth consumption), so it cannot be conducted frequently. As a result, the previous scheduler updates thread priority per very long period as long as 10 million clock cycles. As is shown in figure 4, the memory-intensity tendency changes more frequently than 10 million cycles, so the previous method lacks timeliness and fitness for the priority prediction.

3. Mechanism

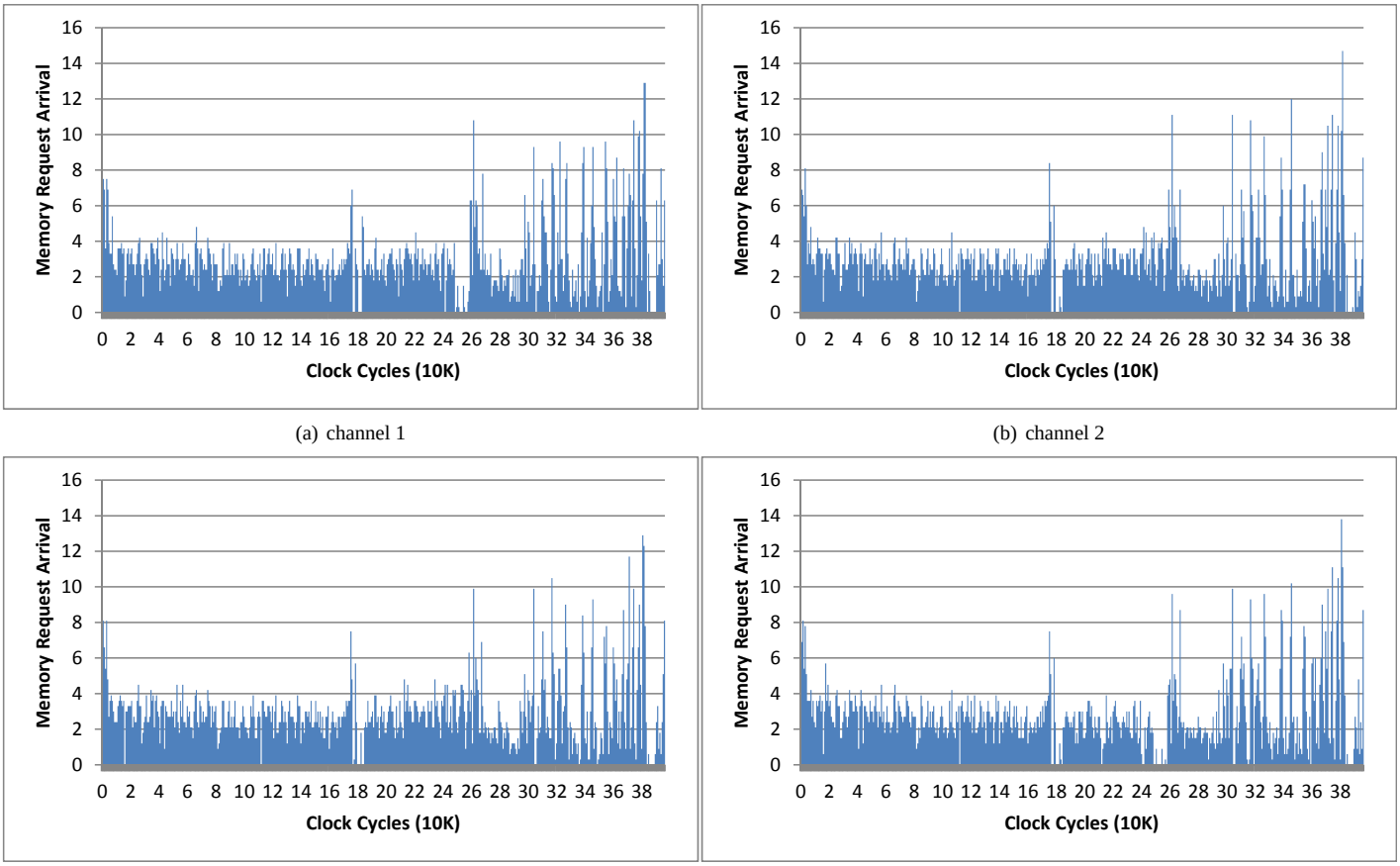
CBM is a fine-grain thread priority scheduling method based on core information. CBM improves the timeliness and accuracy of the priority prediction of Thread Cluster Memory scheduler(TCM, [1]). CBM consists of two separate modules: priority scheduler and memory controller.

CBM places the priority scheduling module on each core (we define the core to which the priority scheduler attached as the ”home core”), not on memory scheduler. Priority scheduler has two registers: the last instruction counter(LIC) and the burst counter(BC). LIC stores the instruction counter value (which is the number of the committed instructions of the home core) of the last LLC miss request derived from the home core. When a new memory request occurs, priority scheduler calculates the distance between the current instruction counter and LIC (We call this distance ”LIC distance”). If LIC distance exceeds *Non-Memory-Intensive Distance Threshold*, the priority scheduler considers that the next memory request is issued after a long calculation period from the last LLC miss. Then, the priority scheduler judges that the home core is in the non-memory-intensive phase.

On the other hand, the BC counts the number of sequential memory requests issued in a short while. If LIC distance is smaller than the *Memory-Intensive Distance Threshold*, BC counts up. If LIC distance exceeds *Non-Memory-Intensive Distance Threshold*, BC is set 0. When the BC value exceeds *Memory-Intensive Burst Threshold*, the priority scheduler considers that the home node is in memory-intensive phase.

When a request misses private cache, priority scheduler predicts memory-intensity of the current thread based on LIC and BC value. Then, priority scheduler adds 1-bit Memory-intensity flag to the memory request. This flag is sent to upper level cache and main memory. As a result, even if there are several LLC modules or several memory channels in the system and the requests from a core are distributed among them, there is no need to gather the statistical information from them. Similar to the Memory-intensity flag, 1-bit LLC-Miss flag is attached to the memory access response back to the private cache. When a private cache miss request causes LLC miss and memory access, LLC-Miss flag is set 1. If the request hits any cache, then LLC-Miss flag is set 0. Therefore, priority scheduler on each core can track LLC miss of each request, and also update LIC and BC registers.

As is mentioned above, the LLC miss detection and LIC update happens when a result of LLC miss access returns to home core. Priority scheduler cannot judge whether in-flight memory



(a) channel 1 (b) channel 2

(c) channel 3 (d) channel 4

Fig. 3 The number of served requests on each channel from Blackscholes workload (with Close Page Policy scheduler, and 15 other threads running simultaneously). We can see that near 250K cycles, the memory request arrival is small in channel 1 and 4, but large in channel 2 and 3. This access pattern happens when the memory requests has high spatial locality and low inter-channel parallelism. In such case, the memory-intensity appears differently between each channel, leading to the wrong memory-intensity judgment and inconsistent thread priority scheduling.

(A) Per-Request		(B) Per-Core	
Resource	Budget	Resource	Budget
Priority Flag	1-bit	LIC Counter	64-bit
		BL Counter	4-bit
Per-Request	1-bit	Per-Core	68-bit

Table 1 Hardware budget count.

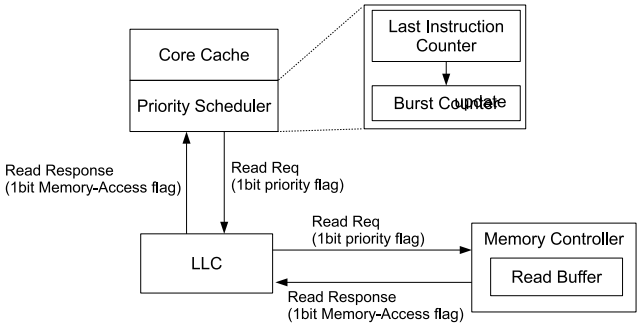


Fig. 5 CBM system structure

requests are the LLC miss requests or not. For this reason, the memory-intensity prediction gets inaccurate to some extent while waiting for any in-flight request. However, the error range due to this factor is as large as the capacity of the ROB of home core at most. The capacity of modern ROB is generally from 32 to 128, so this error range is not so large for the memory-intensity prediction.

4. Implementation

Figure 5 shows the block diagram of CBM scheduler. CBM requires two hardware support: thread-priority scheduler on each core and memory scheduler on each memory channel as described. The major hardware budget is shown in table 1. The required hardware storage cost within priority controller is 68bit per core. The additional hardware cost within memory scheduler part is 1bit per read request.The priority scheduler of each core only needs simple calculation, and the calculation can be conducted by snooping the instruction counter before the memory request happens. Therefore, it does not have a large effect on critical path. Each priority scheduler only needs to calculate the requests from home core, so the calculation time does not increase as the core number increases. The memory scheduler part does not need the meta scheduler to gather all information distributing among all channels, so the calculation load of memory scheduler is smaller than the previous scheduler. For this reason, CBM method has scalability to both the channel number and the core number.

Table 2 Comparison of key metrics on baseline and proposed memory controllers.

Workload	Config	Sum of exec times (10 M cyc)			Max slowdown			EDP (J.s)		
		FR-FCFS	Close	Proposed	FR-FCFS	Close	Proposed	FR-FCFS	Close	Proposed
MTc	1 chan	398	395	374	NA	NA	NA	3.85	3.79	3.79
MTc	4 chan	168	158	153	NA	NA	NA	1.57	1.4	1.31
MTf	1 chan	303	319	305	NA	NA	NA	2.43	2.54	2.38
MTf	4 chan	238	239	236	NA	NA	NA	2.95	2.95	2.89
bl-bl-fr-fr	1 chan	147	145	138	1.18	1.16	1.11	0.48	0.46	0.43
bl-bl-fr-fr	4 chan	78	74	73	1.09	1.03	1.02	0.35	0.31	0.3
c1-c1	1 chan	82	82	79	1.1	1.1	1.05	0.4	0.4	0.36
c1-c1	4 chan	52	46	46	1.06	0.95	0.94	0.44	0.36	0.35
c1-c1-c2-c2	1 chan	230	231	212	1.41	1.43	1.31	1.37	1.39	1.21
c1-c1-c2-c2	4 chan	124	115	112	1.15	1.07	1.05	0.95	0.8	0.77
c2	1 chan	43	43	42	NA	NA	NA	0.36	0.36	0.34
c2	4 chan	30	27	26	NA	NA	NA	0.49	0.4	0.39
c3-c3-c3-c3-c3-c3-c3-c3	4 chan	210	198	193	1.2	1.14	1.11	0.89	0.8	0.75
c4-c4-c5-c5	1 chan	126	127	124	1.08	1.09	1.07	0.37	0.38	0.36
c4-c4-c5-c5	4 chan	71	67	67	1.04	0.98	0.97	0.31	0.27	0.27
fa-fa-fe-fe	1 chan	215	216	200	1.46	1.44	1.32	1.09	1.07	0.92
fa-fa-fe-fe	4 chan	102	96	92	1.17	1.1	1.06	0.6	0.52	0.49
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	282	268	257	1.33	1.25	1.21	1.95	1.73	1.56
fl-fl-sw-sw-c2-c2-fe-fe -bl-bl-fr-fr-c1-c1-st-st	4 chan	618	601	590	1.8	1.73	1.68	4.79	4.45	4.08
fl-sw-c2-c2	1 chan	238	235	220	1.39	1.36	1.25	1.36	1.31	1.14
fl-sw-c2-c2	4 chan	127	119	116	1.11	1.04	1.02	0.95	0.8	0.76
le-le-le-le	1 chan	225	226	210	1.42	1.43	1.33	1.15	1.17	1.03
le-le-le-le	4 chan	167	149	150	1.07	0.95	0.96	1.49	1.17	1.2
li-li	1 chan	109	123	119	0.89	1	0.97	0.74	0.92	0.87
li-li	4 chan	94	74	73	1.29	1.02	1.01	1.49	0.95	0.93
li-li-li-mu-mu-mu-ti-ti	4 chan	590	548	526	2.02	1.83	1.72	7.43	6.56	6.28
li-li-mu-mu	1 chan	378	393	398	1.67	1.6	1.61	3.94	3.83	3.84
li-li-mu-mu	4 chan	215	190	185	1.49	1.27	1.24	2.55	2.02	1.94
st-st-st-st	1 chan	159	156	150	1.24	1.23	1.17	0.55	0.54	0.49
st-st-st-st	4 chan	84	80	78	1.12	1.06	1.04	0.38	0.34	0.32
ti-ti	1 chan	169	162	157	1.26	1.2	1.17	1.84	1.68	1.6
ti-ti	4 chan	102	89	87	1.1	0.96	0.94	1.78	1.38	1.34
Overall		6188	6006	5804	1.27 PFP: 6407	1.21 PFP: 5859	1.17 PFP: 5472	51.43	47.22	44.86

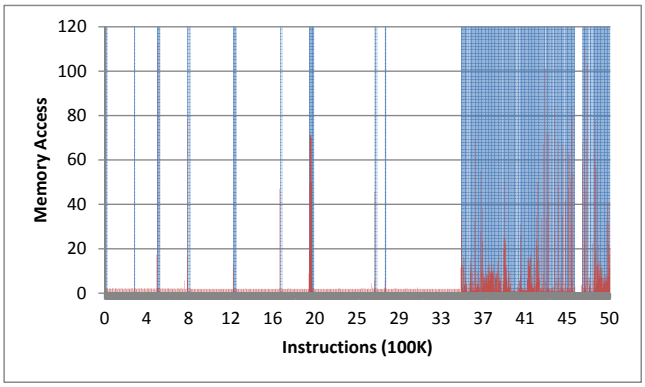


Fig. 6 Priority transition of CBM. The part with blue background is judged as the memory-intensive phase, and the part with white background as the non-memory-intensive phase. CBM can enhance timeliness and accuracy of the memory-intensity recognition from TCM, so the priority transfers appropriately.

5. Evaluation

We implement CBM scheduler on the memory scheduling championship framework[4]. We use three metrics for the evaluation: performance, PFP, and EDP score. We compare CBM with two previously proposed memory scheduler: FR-FCFS, Close Page Policy. Table 2 shows the evaluation result of each scheduler. As is shown in the table 2, CBM outperforms the other schedulers in all metrics. CBM reduces the total execution time by 6.2%, the PFP by 14.6%, and the EDP by 12.8% from the baseline FR-FCFS scheduler.

6. Conclusion and Future Works

We proposed Core Based Memory scheduler (CBM) in this paper. CBM separates the priority scheduler module from memory controller, and places it on each core. By this separation, CBM does not need heavy inter-channel communication that was conducted periodically in existing memory scheduling algorithms to gather the memory request statistics. As a result, CBM can update memory-intensity information every memory access, leading to finer-grain thread priority update. Therefore, CBM enhances the timeliness and accuracy of thread priority prediction simultaneously. The priority scheduler on CBM is placed on each core and requires no meta scheduler gathering all statistical information, so it is scalable to the overall core number and the memory channel number in the system. Moreover, memory schedulers in all memory channels can serve memory requests from a core synchronously (or, with the same priority) without inter-channel communication. For this reason, all memory channels can cooperate on CBM scheduler efficiently.

We evaluated CBM scheduler by using the memory scheduling championship framework and its workloads. The experimental

result showed that CBM scheduler improves the total execution time by 6.2%, the PFP by 14.6%, and the EDP by 12.8% from the baseline FR-FCFS scheduler respectively.

CBM method can be also applied to the case that the transfer of memory requests will be irregularly delayed, such as Network on Chip(NoC) structure. The request priority is set by the priority scheduler on each core, so all requests have already known their own priority when they are issued from home core. Therefore, these priority information can be used for the routing algorithm of NoC for example. Combining CBM scheduling method with the NoC routing algorithm is our future works.

References

[1] Kim, Y., Papamichael, M., Mutlu, O.: Thread cluster memory scheduling: Exploiting differences in memory access behavior, *Micro*, 43rd (2010)
[2] Mutlu, O., Moscibroda, T.: Parallelism-Aware Batch Scheduling : Enhancing both Performance and Fairness of Shared DRAM Systems, *ACM SIGARCH Computer Architecture News*, pp. 1–12 (2008).
[3] Kim, Y., Han, D., Mutlu, O.: ATLAS : A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers, *High Performance Computer Architecture (HPCA)-16*, (2010).
[4] Chatterjee, N., Balasubramonian, R.: USIMM: the Utah SImulated Memory Module, *3rd JILP Memory Scheduling Championship (MSC)*, (2012).