Service Value Aware Memory Scheduler by Estimating Request Weight and Using per-Thread Traffic Lights

Keisuke Kuroyanagi, André Seznec

Service Value Aware Memory Scheduler by Estimating Request Weight and Using per-Thread Traffic Lights *

Keisuke Kuroyanagi
INRIA/IRISA/The University of Tokyo
ksk9687@is.s.u-tokyo.ac.jp

André Seznec
INRIA/IRISA
seznec@irisa.fr

ABSTRACT

The memory controller has become one of the performance enabler of a computer system. Its impact is even higher on multicores than it was on uniprocessor systems.

In this paper, we propose the sErvice Value Aware memory scheduler (EVA) to enhance memory usage. EVA builds on two concepts, the request weight and the per-thread traffic light. For a read request on memory, the request weight is an evaluation of the work allowed by the request. Per-thread traffic lights are used to track whether or not in a given situation it is worth to service requests from a thread, e.g. if a given thread is blocked by refreshing on a rank then it is not worth to serve requests from the same thread on another rank.

The EVA scheduler bases its scheduling decision on a service value which is heuristically computed using the request weight and per-thread traffic lights. Our EVA scheduler implementation relies on several hardware mechanisms, a request weight estimator, per-thread traffic estimators and a next row predictor. Using these components, our EVA scheduler estimates scores to issue scheduling decisions.

EVA is shown to perform efficiently and fairly compared with previous proposed memory schedulers. For the memory scheduling competition, we submit EVA to Delay and PFP tracks and EVA-E (EVA with optimisations for EDP) to EDP track. Total delay is 2975 $\times 10^7$, EDP value is 19.79 and PFP value is 2842 $\times 10^7$.

1. INTRODUCTION

The performance impact of scheduling requests on main memory has been increasing over the last few years especially with the advent of multicores and multithreaded processors. Smart memory controllers scheduling policies have now become of major importance.

Several memory scheduling policies have been recently proposed. Some of these methods propose prediction strategy [1, 2]. They predict whether an opened row should be closed or not, which row should be speculatively activated and so on. These methods mainly focused on improving the row hit rate in order to minimize accesses latency and maximize throughput. However, requests issued from multiple threads disturb the predictability of the tracked phenomena and induce a very complex request behavior. To effectively handle multiple cores and threads, some recent methods are thread aware [3, 4]. These methods try to maximize memory throughput while not impairing fairness or vice-versa.

Our proposition, the sErvice Value Aware memory scheduler (EVA), takes scheduling decisions based on an evaluation of the potential performance benefit or cost of servicing the requests. It computes this service value, based on two concepts, the request weight and the per-thread traffic light. For a read request on memory, the request weight is an evaluation of the quantity of work before the next read request. Per-thread traffic lights are used to track whether or not in a given situation it is worth to service requests from a thread, e.g. if some request is blocked for a given thread then it is not worth to serve requests from the same thread even when possible. Additionally, EVA uses per-thread next row prediction for speculative activation and some other advanced techniques. Our experiments indicate that EVA performs efficiently and fairly compared with previous proposed memory schedulers.

The remainder of the paper is organized as follows. First, we present a design overview of our EVA memory scheduler in Section 2, including the request weight estimator, the per-traffic lights and the next row address predictor. Section 3 describes the EVA memory scheduler algorithm. Section 4 introduces the variations of the scheduler targeting the different tracks of the memory scheduling competition. Section 5 presents evaluation results. Finally, Section 6 concludes this paper.

2. DESIGN

2.1 Overview

Figure 1 illustrates the general principle of the EVA memory scheduler. EVA uses a request weight estimator, per-thread traffic lights and a next row predictor. These components are described below. Information provided by these components are used by the scheduler to decide which command should be issued next.

2.2 Request weight estimation

2.2.1 General principle

For deciding that a request should be issued or not, we rely on the new concept, called request weight.

For a request R, we define its request weight as the quantify of useful work from the relevant thread that can

*This work was partially supported by the European Research Council Advanced Grant DAL No 267175 and the Study Program at Overseas Universities, the Graduate school of Information Science and Technology from the University of Tokyo.

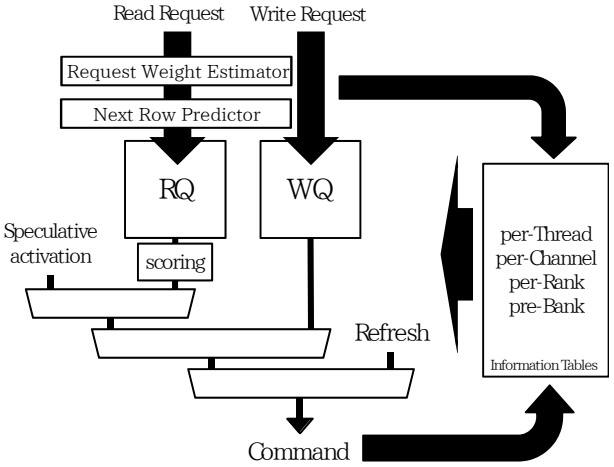

Figure 1: Overview of EVA memory scheduler.

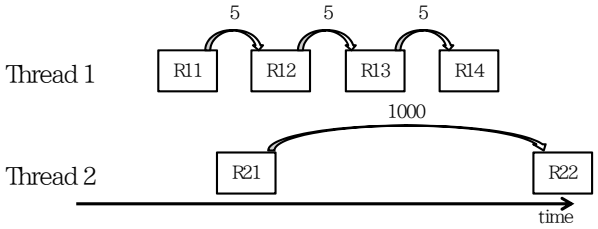

Figure 2: Basic idea of request weight.

be executed after servicing R and without servicing any subsequent request from the same thread.

The example in Figure 2 illustrates our general intuition: six read requests R11-R14 for Thread 1, and R21-R22 for Thread 2 R12, R13, R14 arrived 5 cycles after the prior request while R21 arrived between R11 and R12, and R22 arrived 1000 cycles after R21. In this situation, R21 should be given priority even if R11-R14 are row hit requests because it allows to trigger more useful work than servicing R12.

Unfortunately the request weight cannot be precisely known, even at runtime. Therefore it has to be estimated. In our proposal, we differentiate two situations 1) If at run-time, when the next memory read request from the same thread after request R arrives after servicing R then we use the number of cycles after this servicing as the request weight for R. 2) Otherwise, i.e. R is still waiting for execution, the weight is calculated as the number of instructions separating R from the following memory read request (this number is available from the instruction sequence number in ROB).

Some previous proposals like ATLAS [3] or Thread Cluster Memory Scheduling [4] change the thread priority depending on the behaviour of each thread. However, EVA tracks the behaviour of each individual request and changes thread priority more quickly. Our weight estimation is independent from the behavior of the other threads and it can be computed easily provided that instruction sequence numbers and thread numbers are forwarded up to the memory scheduler.

2.2.2 *Request weight estimator*

To utilize request weight estimation for scheduling, this estimation must be predicted in most cases, i.e. when the subsequent request has not been already sent by the processor. Estimated request weights are provided by the request weight estimator that is directly derived from the ITTAGE [5] predictor, a state-of-the-art indirect branch predictor.

Table 1: Configuration of request weight estimator.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
#entries	2^{12}	2^{11}		2^{11}			2^{10}		2^{10}	
tag bits	10	11		13			15		17	
pc hist	0	0	1	2	3	5	9	16	32	64
row hist	0	0	1	2	3	5	8	8	8	8

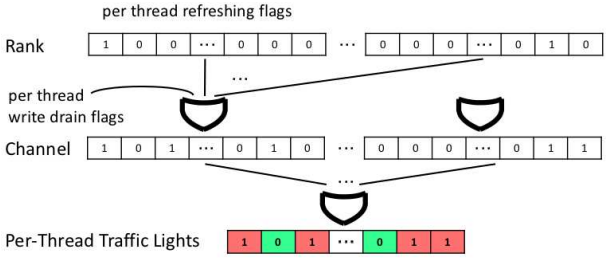

Figure 3: Updating method of per-thread traffic lights.

Table 2 shows the configuration of this estimator. 12-bit entry values are used as request weight, the maximum value (MAX_VALUE) means equal to or greater than that value. We use ten logical tables T_{0-9}. Among them, $T_{1,2}$, T_{3-5}, $T_{6,7}$, $T_{8,9}$ are severally sharing one physical table, that is to say, there are five physical prediction tables in this estimator. As history , we use 1 bit from the program counter and 1 bit from the DRAM row number and manage them as per thread PC and row histories. Note that we use relative short histories to accesses T_{0-5}, the main objective being to increase the associativity of the estimator. On the other hand, longer histories are used to access T_{7-9} to track request patterns that change their behavior depending on the execution history. There is a much larger number of hard-to-predict behaviors than for branch prediction. Therefore, we employ blacklist filtering method [6] to track such requests and to avoid useless updating. This filtering consists of 64-entry 16-way set associative table.

The weight estimator provides the predicted request weight as soon as the memory request arrives. The estimated value is overwritten by small weight if its confidence is low to keep away from falsely predicting large weights. This value is stored in the read request queue. After the next request arrives, first, the actual request weight is calculated as mentioned above. Then, the request weight estimator is updated with the calculated weight. Predicting precise weight is difficult and useless; therefore, we consider the estimation as correct when $actual\_value * 0.5 - 1 < estimated\_value < actual\_value * 1.5 + 1$ and we update prediction table with $(estimated\_value + actual\_value) \div 2$.

2.3 Per-thread traffic lights

On a multicore, requests from the different threads interfere. In particular, a request is unlikely to allow substantial progress on the execution of a thread if a prior

Table 2: Configuration of next row predictor.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
#entries	2^{10}	2^{10}		2^{10}			2^{10}	
tag bits	10	11		13			15	
pc hist	0	0	1	2	3	5	8	8
row hist	0	0	1	2	3	5	16	32

request from a thread is blocked by a rank refreshing or the channel servicing write memory requests.

To track such situations, we introduce the per-thread traffic light. A per-thread traffic light is a one-bit flag that indicates whether requests from the thread are worth servicing or not.

This status is updated at the beginning of every DRAM cycle as shown in Figure 3. First, for all threads, EVA checks if there are read requests that are related to a refreshing rank and raise a refreshing flag for such threads. Then, EVA checks if there are read requests that are related to write draining channel and raise a write drain flag for such threads. Finally, the ORed value of all per-thread flags is used as the per-thread traffic lights. A raised flag, which indicates the light is red, means that processing read requests from the thread is useless. EVA changes scheduling priority based on per-thread traffic lights to maximize execution performance. Details of scheduling strategy are described in Section 3.

2.4 Next row prediction

For speculative activation, EVA implements a next row prediction method that predicts the channel, the rank, the bank and the row of next read request. In our proposal, EVA uses a per-thread next row prediction method based on ITTAGE. Using an ITTAGE derived next row predictor can be effective, essentially because a row features a large number of memory blocks.

For accessing the tables of the next row predictor, we privilege row history on PC history because the behavior of target address is tend to be related to memory address stream rather than to instruction address stream. Predicted rows with high confidence are used for speculative activation. It should be noted that memory controller often have to handle requests from multiple threads at the same time; this makes it difficult to decide which row should be opened. To deal with this issue, we use information of request weight and traffic lights to choose which row should be activated. Details of speculative activation method are described in Section 3.4.

2.5 Bus utilization tracking

Request weight sometimes cannot be well estimated because of initial misses, unpredictability behavior or capability limitation of the estimator. In such situations, performance and fairness can be greatly impaired.

To manage these issues, EVA tracks bus utilization for each thread and uses it as a global index of priority. We use the following heuristic similar to the bandwidth allocation method proposed by J.Turner [7]. Bus utilization is measured by employing per-thread tickets. Every $NUM\_CORES \times T\_DATA\_TRANS \times 6$ cycles, each thread get one ticket and when read request is issued, the thread pays one ticket. EVA decides request priority also depending on how many tickets the thread has.

2.6 Information used for scheduling

EVA maintains some information for each thread, each channel, each rank and each bank.

For each thread, we maintain a read request counter, a thread weight, an oldest request pointer, a latest request pointer, a traffic light, ticket numbers, information for request weight estimator and information for next row predictor. The thread weight is the sum of weights of the waiting requests from the thread.

For each channel, we maintain a per thread read request counter, a 2-bit counter that indicates write drain state, and a last refreshed rank.

For each rank, we maintain a read request counter, a write request counter, a per thread read request counter, a rank weight, a activation history and a last refresh cycle register. Rank weight is sum of the request weights that are related to the rank. The activation history consists of four counters that record the cycle value when activation commands are issued to track how many activations are issued in $tFAW$. The last refresh cycle register records the cycle value that last refresh command was issued.

For banks, we maintain a 2-bit preactivated flags and a read request counter.

2.6.1 Managing the information

This information are updated after the DRAM commands or receiving a new request as follows:

receiving read request
: Read request counters are updated. Thread weight and rank weight are updated by adding estimated request weight. If a prior request is still queueing, these values are adjusted with computed actual request weight. Oldest and latest pointers are updated with channel and queue entry number of the new read request when needed.

receiving write request
: Write request counters are updated.

issuing column read command
: Read request counters are updated. Thread weight and rank weight are updated by subtracting the request weight of issuing request. Oldest pointer for the thread is updated with the second oldest request. Tickets counter of the related thread is decremented.

issuing column write command
: Write request counters are updated.

issuing activation command
: The oldest entry of activation history for the rank is replaced with current cycle value.

issuing refresh command
: Last refresh cycle of target rank is replaced with current cycle value.

3. SCHEDULING METHOD

We assume that the memory scheduler enforces the refreshing on deadlines if needed as well as it enforces the respect of the semantics, i.e. if a read request addresses a block to be written then the write is performed before the read.

Table 3: Priority order of DRAM commands.

	command	condition
1	refresh	$!DRAIN\_STRONG$
2	priority column read	
3	column write	$DRAIN\_STRONG$
4	priority activation and precharge	
5	column read	
6	column write	
7	command for write requests	$!NOT\_DRAIN$
8	activation and precharge for read	
9	activation and precharge for write	
10	precharge unrefferd row	
11	speculative activation	

Apart these cases, EVA estimates if DRAM commands is worth to be launched. Then to select the issued command, EVA uses the priority order illustrated in Table 3 Priority requests are decided as shown in Algorithm 1. Below, we describe our algorithms to estimate whether commands are worth to be launched or not.

Algorithm 1 Deciding priority read requests.

```
if thread_value ≥ MAX_VALUE then
  priority ← true
end if
if tickets ≥ MAX_NUM_TICKETS − 8 then
  priority ← true
end if
```

3.1 Write requests

On every cycle, EVA checks whether each channel should switch to the write drain mode through Algorithm 2. Three water marks(HI_WM, MD_WM, LO_WM) and three write drain levels ($DRAIN\_STRONG$, $DRAIN\_WEAK$, NOT_DRAIN) are considered. $DRAIN\_WEAK$ is a write drain mode that can be interrupted by any refreshing or priority request. When there are no rows referred by more than three requests, EVA exits $DRAIN\_WEAK$ mode.

Once EVA gets into the write drain mode, it follows the simple way to process write requests until it leaves from the write drain mode. First, EVA tries to serve the row hit requests. Then, EVA opens the row that is demanded by the largest number of write requests.

Algorithm 2 Checking write drain mode

```
HI_WM ← WQ_CAPACITY − 5
MID_WM ← HI_WM − 8
LO_WM ← MID_WM − 8
if write_drain ≠ NOT_DRAIN then
  if write_queue_length > MID_WM then
    write_drain ← DRAIN_STRONG
  else if write_queue_length > LO_WM then
    if interuppted_by_valuable_read() then
      write_drain ← NOT_DRAIN
    else if max_reqs_row ≤ 2 then
      write_drain ← NOT_DRAIN
    else
      write_drain ← DRAIN_WEAK
    end if
  else
    write_drain ← NOT_DRAIN
  end if
end if
if write_queue_length > HI_WM then
  write_drain ← DRAIN_STRONG
end if
```

3.2 Refreshing

Algorithm 3 Refresh

```
num ← num_issued_refreshes
rest ← next_refresh_completion_deadline − CYCLE_VAL
if num * 2 < rest ÷ T_REFI then
  refresh = false
else if write_drain = DRAIN_STRONG then
  refresh = false
else if rank.write_count > HI_WM/NUM_RANKS then
  refresh = false
else if rank.read_count = 0 then
  refresh = true
else if rank_has_high_prio_thread() then
  refresh = false
else
  refresh = true
end if
```

The EVA scheduler tries to issue refresh commands in advance at intervals of $T\_REFI \div 2$ cycles. However, refreshes should not be given always priority. In some cases, the scheduler had better wait for requests even though the rank has nothing to do. First, we try to distribute the refresh over the whole refreshing interval. Second, write draining has priority over refresh. Third, if the number of writes to the rank is high then refresh is delayed in order to allow future write draining and finally if there is a high priority thread, refresh should also be delayed.

Algorithm 3 illustrates the precise algorithm implemented in our proposal.

Algorithm 4 Scoring read requests

```
score ← 0
if traffic_light = GREEN then
  score ← 4
end if
if is_from_max_value_thread() then
  score ← score + 4
end if
if is_oldest_in_thread() then
  score ← score + 1
end if
if thread.read_count = 1 then
  score ← score + 2
end if
```

3.3 Read requests

EVA tries to serve read request on every cycle unless the channel is in the write drain mode or a refresh command is issued.

First, EVA computes a score for each issuable request as illustrated in Algorithm 4. Each item in the algorithm favors some particularity of the request: the thread with GREEN traffic light, the thread with the maximum sum of request weights, oldest request in the thread and only one remaining request from the thread.

Based on this score, EVA looks for the candidate column read, activate and precharge command that has the highest score among the same type commands. Among the candidate commands, EVA issues the one with the highest score. When several candidates have same score, activate commands take precedence over precharge commands. If there are no more requests for the accessed row then EVA issues an auto precharge command at the same time as dispatching a column read command.

3.4 Speculative activation

Algorithm 5 determines which row EVA will be activated speculatively among the next predicted rows with high confidence.

First, the thread status is checked in order to determine if the traffic light is green and if there are not more than two pending requests, because it would take a long time to serve the next request if the traffic light is red or if many requests are pending. Then, the rank status is checked to determine if there are not any requests for the rank and if the rank is not preactivated.

In addition, EVA tests the constraint on parameter tFAW. At most four activation commands can be issued in a tFAW interval. Therefore as speculative activation has very low priority, EVA only launches speculative activation if at most two activation commands are already inflight in that interval. Then for each candidate, EVA computes a score based on the last request value of the thread and the thread read request counter, then EVA chooses the address that has the lowest score, which is the most likely to be used first. The selected row is speculatively activated if the value of $PREACTCTR$, which is signed 6-bit saturated counter, is higher than the threshold. This counter shows global confidence of speculative activation. $PREACTCTR$ is incremented if the selected row is used; otherwise, three is subtracted from this counter value.

Algorithm 5 Detecting what row should be activated speculatively

```
candidates ← ∅
for all t ∈ threads do
  (channel, rank, bank, row) ← t.pred_next_addr
  if t.traffic_light = GREEN&&t.read_count ≤ 2 then
    if rank.read_count = 0&&!bank.pre_activated then
      if num_act_in_tACT(channel, rank) < 3 then
        candidates ← t
      end if
    end if
  end if
end for
preact ← arg max_{t∈candidates} t.oldest.value + t.read_count × 100
if PREACTCTR ≥ 28 then
  activate(preact.addr)
end if
```

4. EVA FOR THE MEMORY SCHEDULING CHAMPIONSHIP

The EVA scheduler described in the previous sections is submitted to both the performance track and fairness track. By construction, the EVA scheduler tries to favor the requests with high quantity of work through the use of the request weight estimator, while the use of bus utilisation tracking enables some fairness in the access to the DRAM.

For the energy-delay product track, we present a slightly modified version of EVA, that targets the energy-product metric that will be used to rank the proposals at the competition.

4.1 Energy-Delay Product track

DRAM power consumption is a limited part of the power dissipated by the system. Therefore our experiments showed that instead of trying to manage power in the DRAM, focusing on minimizing the execution time of the slowest thread was the best strategy to minimize the energy-delay product metric.

We use an extra counter for each thread to monitor the execution progress by adding all the request weights. EVA use this value to compute read request scores and deduces request weights. We call this scheduler EVA-E (EVA with "optimizations" for Energy-delay product track).

Table 4: Storage budgets for all the components

Component	Configuration	Budget
Request weight estimator	276,480-bit prediction tables 57,344-bit blacklist filter 64-bit, 8-bit hist × 16-thread 8-bit and 4-bit counters	334,988
Next row predictor	168,960-bit prediction tables 32-bit, 8-bit hist × 16-thread 8-bit and 4-bit counters	169,612
Write row count table	(2+1+3+17)-bit tag, 8-bit cntr × 128-entry × 4-ch	15,872
Thread info table	8-bit read counter 13-bit weight counter (2+8)-bit pointer × 2 1-bit traffic light 7-bit tickets counter 67-bit for request value est. 59-bit for next row pred. 160 × 16-thread	2560
Channel info table	2-bit write drain state 1-bit last refresh rank 8-bit read count × 16-thread 131 × 4-ch	524
Rank info table	8-bit read counter 8-bit read counter × 16-thread 8-bit write counter 13-bit rank weight counter 64×4-bit act history 64-bit last refresh cycle 477 × 4-ch × 2-rank	3,816
Bank info table	8-bit read counter 17-bit preactivated row 2-bit preactivate flags 27 × 4-ch × 2-rank × 8-bank	1,728
Others	128-bit for RNG (Xorshift [8]) 6-bit $PREACTCTR$ 12-bit request weight in RQ × 255-entry × 4-ch	128 6 12,240
Total		541,474

4.2 Storage Budget

Table 4 illustrates the storage budget for the submitted EVA memory scheduler. For this count, we assume a maximum of 16 threads and a maximum read queue length of 255 entries.

The total storage budget size for EVA is 541,474 bits and EVA-E additionally uses per thread 64-bit counters; thus, the total budget size of EVA-E is 542,498 bits. Both of them are less than 557,056 bits (68KB).

5. RESULTS

We have evaluated the EVA memory scheduler with the distributed framework [9]. In this section, EVA refers to the memory scheduler for Delay and Performance-Fairness Product track and EVA-E refers to the memory scheduler for Energy-Delay Product track.

Table 5 illustrate our simulation results. For the total delay metric, EVA achieves 2975 $\times 10^7$. For the PFP metric, EVA achieves 2842 $\times 10^7$. EVA-E achieves 19.79 for the EDP

Workload	Config	Sum of exec times (10 M cyc)			Max slowdown			EDP (J.s)		
		FCFS	Close	EVA	FCFS	Close	EVA	FCFS	Close	EVA-E
MT-canneal	1 chan	418	404	363	NA	NA	NA	4.23	3.98	3.57
MT-canneal	4 chan	179	167	165	NA	NA	NA	1.78	1.56	1.55
bl-bl-fr-fr	1 chan	149	147	138	1.20	1.18	1.12	0.50	0.48	0.42
bl-bl-fr-fr	4 chan	80	76	76	1.11	1.05	1.06	0.36	0.32	0.32
c1-c1	1 chan	83	83	78	1.12	1.11	1.07	0.41	0.40	0.37
c1-c1	4 chan	51	46	48	1.05	0.95	1.00	0.44	0.36	0.40
c1-c1-c2-c2	1 chan	242	236	213	1.48	1.46	1.35	1.52	1.44	1.25
c1-c1-c2-c2	4 chan	127	118	119	1.18	1.10	1.12	1.00	0.85	0.89
c2	1 chan	44	43	43	NA	NA	NA	0.38	0.37	0.37
c2	4 chan	30	27	28	NA	NA	NA	0.50	0.39	0.44
fa-fa-fe-fe	1 chan	228	224	202	1.52	1.48	1.39	1.19	1.14	0.97
fa-fa-fe-fe	4 chan	106	99	98	1.22	1.15	1.13	0.64	0.56	0.56
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	295	279	262	1.40	1.31	1.24	2.14	1.88	1.68
fl-fl-sw-sw-c2-c2-fe-fe--bl-bl-fr-fr-c1-c1-st-st	4 chan	651	620	569	1.90	1.80	1.65	5.31	4.76	4.05
fl-sw-c2-c2	1 chan	249	244	221	1.48	1.43	1.30	1.52	1.44	1.23
fl-sw-c2-c2	4 chan	130	121	123	1.13	1.06	1.07	0.99	0.83	0.88
st-st-st-st	1 chan	162	159	147	1.28	1.25	1.18	0.58	0.56	0.49
st-st-st-st	4 chan	86	81	81	1.14	1.08	1.09	0.39	0.35	0.35
Overall		3312	3173	2975	1.30 PFP: 3438	1.24 PFP: 3149	1.20 PFP: 2842	23.88	21.70	19.79

Table 5: Comparison of key metrics on baseline and proposed schedulers. c1 and c2 represent commercial transaction-processing workloads, MT-canneal is a 4-threaded version of canneal, and the rest are single-threaded PARSEC programs. "Close" represents an opportunistic close-page policy that precharges inactive banks during idle cycles.

metric. These are respectively 10.2%, 17.3% and 17.1 % lower than the scores of FCFS scheduler and 6.2%, 9.7% and 8.8 % lower than the scores of close-page policy scheduler.

6. CONCLUSION

We have proposed a new structure of DRAM memory scheduler, the sErvice Value Aware memory scheduler EVA. EVA relies on two concepts, the request weight, i.e. the quantity of work that a request enables, and per-thread traffic lights which evaluate whether a thread is blocked by a pending request. EVA predicts request weights and also relies on advanced speculative row selection.

The scheduling decision on EVA is based on a fixed priority to select among the launchable commands. The launchable commands are determined through 3 relatively simple algorithms involving no loops: a hardware implementation of these algorithms should be very effective.

Our experiments shows that our EVA proposal outperform the FCFS scheduler and the close-page policy scheduler on the three metrics proposed for the Championship.

7. REFERENCES

[1] Vladimir V. Stankovic and Nebojsa Z. Milenkovic. Dram controller with a complete predictor. *IEICE Transactions*, 92-D(4):584–593, 2009.

[2] Ying Xu, Aabhas S. Agarwal, and Brian T. Davis. Prediction in dynamic sdram controller policies. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 128–138, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, 2010.

[4] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society.

[5] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction Level Parallelism*, 8, Febrary 2006.

[6] Yasuo. Ishii, Keisuke Kuroyanagi, Takeo Sawada, Mary Inaba, and Kei Hiraki. Revisiting Local History to Improve the Fused Two-Level Branch Predictor. *2nd JILP Workshop on Computer Architecture Competitions*, 2011.

[7] J. Turner. New directions in communications (or which way to the information age?). *Communications Magazine, IEEE*, 24(10):8–15, January 2003.

[8] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[9] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the Utah SImulated Memory Module. Technical report, University of Utah, 2012. UUCS-12-002.