Adaptive Compute-phase Prediction and Thread Prioritization to Mitigate Memory Access Latency

Ismail Akturk
Department of Electrical and Computer Engineering
University of Minnesota
MN 55411, USA
aktur002@umn.edu

Ozcan Ozturk
Department of Computer Engineering
Bilkent University
Ankara 06800, Turkey
ozturk@cs.bilkent.edu.tr

ABSTRACT

The full potential of chip multiprocessors remains unexploited due to the thread oblivious memory access schedulers used in off-chip main memory controllers. This is especially pronounced in embedded systems due to limitations in memory. We propose an adaptive compute-phase prediction and thread prioritization algorithm for memory access scheduling for embedded chip multiprocessors. The proposed algorithm efficiently categorize threads based on execution characteristics and provides fine-grained prioritization that allows to differentiate threads and prioritize their memory access requests accordingly. The threads in compute phase are prioritized among the threads in memory phase. Furthermore, the threads in compute phase are prioritized among themselves based on the potential of making more progress in their execution. Compared to the prior works First-Ready First-Come First-Serve (FR-FCFS) and Compute-phase Prediction with Writeback-Refresh Overlap (CP-WO), the proposed algorithm reduces the execution time of the generated workloads up to 23.6% and 12.9%, respectively.

1. INTRODUCTION

In response to increased pressure on memory subsystem due to the memory requests generated by multiple threads, an efficient memory access scheduler has to fulfill the following goals:

- serve memory requests in a way that cores are kept as busy as possible
- organize the requests in a way that the memory bus idle-time is reduced

These goals are much more critical for embedded systems due to limitations in memory. One approach to keep cores as busy as possible is to categorize and prioritize threads based on their memory requirements. Threads can be categorized into two groups: memory-non-intensive (i.e., threads in compute phase), and memory-intensive (i.e., threads in memory phase). Kim et al. [5] proposed a memory access scheduler that gives higher priority to memory-non-intensive threads, and gives lower priority to memory-intensive threads. The reason behind such prioritization is that memory-non-intensive threads (i.e., threads in compute phase) can make fast progress in their executions, so the cores can be kept busy. On the other hand, memory-intensive threads (i.e., threads in memory phase) have more memory operations and they do not use computing resources as often.

Ishii et al. followed the same idea of prioritizing the threads based on their memory access requirements. They enhanced prioritization mechanism with a fine-grained priority prediction method. This fine-grained priority prediction method is based on saturation counters [4]. They do not rely on time quantum (typically some millions of cycles) to categorize threads as memory-non-intensive and memory-intensive, instead they employ saturation counters to categorize threads on the fly. In addition, they proposed writeback-refresh overlap that reduces memory bus idle-time. Writeback-refresh issues pending write commands of the ranks that are not refreshing and refreshes a given rank concurrently. This means that the issuing write commands (of rank that is not refreshing) and refreshing a rank are overlapped. This reduces the idle time of the memory bus and enhances the performance of the memory subsystem.

MES14 June 15 2014, Minneapolis, MN, USA
Copyright 2014 ACM 978-1-4503-2822-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2613908.2613919.

2. PROBLEM STATEMENT

The necessity of distinguishing threads based on their memory access requirements is well understood and many research efforts have exploited this fact. Kim et al. [5] and Ishii et al. [4] provided examples of thread classification and prioritization mechanisms. They categorize threads into two groups, namely memory-non-intensive (i.e., threads in compute phase), and memory-intensive (i.e., threads in memory phase). Although they distinguish threads into different groups, they do not differentiate the threads in the same group. We believe that fine-grained prioritization is required even for the threads in the same group (i.e., memory-non-intensive or memory-intensive) to maximize the overall system performance and utilize the memory subsystem at the highest degree. For this reason, we introduce a fine-grained thread prioritization scheme that can be employed by existing state-of-the-art memory access schedulers.

In addition to that, the thread classification scheme presented in the work of Ishii et al. [4] is based on saturation counters. Saturation counters provide effective metrics to understand threads to be in compute phase or in memory phase. In determination of this, interval and distance thresholds are used. These thresholds are predefined and determined empirically. Although they are effective, they are vulnerable to short distortions and bursts that may result in wrong classification of threads. We believe that these thresholds have to be updated appropriately depending on the execution characteristics of the threads to classify them with higher accuracy. For this reason, we enhanced phase prediction scheme of Ishii et al. and make it adaptive.

3. MOTIVATION

The classification of threads running on chip multiprocessors is essential to improve memory subsystem performance. Since the execution characteristics of the threads may change during their lifetime, such a classification has to be updated accordingly. Mainly, a thread can be either in compute phase, or memory phase for a given time of its execution. For this reason, the detection of a phase that a thread is currently in and prediction of the phase that a thread is going to be in have significant importance in scheduling memory accesses. There has to be a memory access scheduler that can predict the execution phases of threads efficiently, and prioritize them to access memory. The prioritization has to be fine-grained and the phase detection has to be accurate, thereby motivating us to implement fine-grained prioritization and adaptive phase prediction in memory access scheduler.

4. ADAPTIVE COMPUTE-PHASE PREDICTION

Ishii et al. [4] proposed a memory access scheduling algorithm that we call Compute Phase Prediction with Writeback-Refresh Overlap (CP-WO). Compute Phase Prediction with Writeback-Refresh Overlap scheduler can predict the execution phase of a thread on the fly. Typically there are two phases a thread may be in. A thread may be either in compute phase, or in memory phase. The threads in compute phase are memory-non-intensive. On the other hand, the threads in memory phase are memory-intensive. The threads in compute phase are given higher priorities for memory accesses. The reason behind this is that the threads in compute phase can make fast progresses and keep cores busy, thereby improving the performance and enhancing the utilization. On the other hand, the threads in memory phase spend more time on memory and have less computation, thus leave cores idle. For this reason, the threads in compute phase are prioritized.

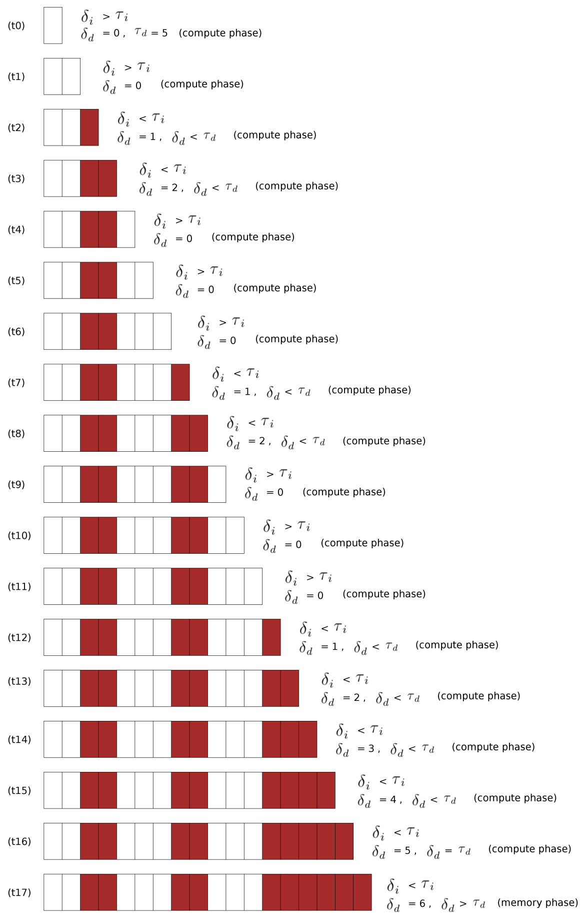
The idea of prioritizing the threads in compute phase (i.e., memory-non-intensive) is also used by Kim et al. [5] in their thread cluster memory (TCM) scheduler. The thread cluster memory scheduler classifies threads into two groups, namely, memory-non-intensive threads and memory-intensive threads. It prioritizes the memory access requests of memory-non-intensive threads over memory-intensive threads. The prioritized memory-non-intensive threads will spend less amount of time on memory operations and return back to the execution much earlier. This way, cores in an embedded chip multiprocessor can be kept as busy as possible, increasing the throughput and improving the performance.

The thread cluster memory scheduler classifies threads whenever the time quantum exceeds a certain threshold, typically in the range of million cycles. Threads are classified at the beginning of each quantum based on memory access patterns. Due to the dynamic behavior of threads, their execution characteristics (i.e., memory access pattern) may change before the time quantum expires. If this is the case, threads have to be re-clustered to properly prioritize them. However, the thread cluster memory scheduler does not have capability of re-clustering threads before the time quantum expires. Threads are treated as they started, although they may change their execution phase, which in turn, requires adjustments in priorities of threads. For this reason, the thread cluster memory scheduler can not respond to the changes in memory access patterns of threads in a timely manner. Due to this limitation, it blindly misses possible improvements on performance and fairness.

To overcome the barrier in thread cluster memory scheduler, Ishii et al. employed saturation counters to classify threads. Saturation counters help to respond to changes in memory access pattern of a thread in a faster manner compared to time quantum approach of thread cluster memory scheduler. Another difference between thread cluster memory scheduler and the scheduler of Ishii et al. is that the former uses memory traffic generated by L2 cache miss to cluster threads, while the latter uses the committed number of instructions to cluster threads. We believe that the former provides better indication of threads being in compute phase, or in memory phase.

Ishii et al. used saturation counters to determine if a thread is in compute phase or in memory phase. These saturation counters are *interval counter* and *distance counter*. The interval counter specifies the number of committed instructions between the last two cache misses for a thread. If an interval counter (i.e., δ_i) of a thread exceeds the interval threshold (i.e., τ_i), then thread is predicted to be in compute phase and the distance counter (i.e., δ_d) is set to zero. On the other hand, the distance counter is incremented if the interval counter stays below the interval threshold. If there are consecutive accesses whose interval counter stays below the interval threshold that leads distance counter to exceed the distance threshold (i.e., τ_d), then a thread is considered to be in memory phase. The distance threshold determines how long a thread is going to be treated as it is in compute phase; although, it does not satisfy the interval counter constraint. This allows tolerating short distortions and small bursts that may be seen in compute phase and thereby, not treating a thread to be in memory phase, immediately. However, it is important to decide how long to tolerate a thread that does not satisfy the interval counter constraint before considering it to be in memory phase and vice versa.

The distance threshold (τ_d) and the interval threshold (τ_i) are predefined in the original work. The higher distance threshold becomes inappropriate for most of the cases since it keeps a thread in compute phase longer; although the thread actually is in memory phase. On the other hand, smaller distance threshold makes a thread vulnerable to short distortions and bursts, so a thread is treated as it is in memory phase; although, it is in compute phase. To deal with such anomalies, we introduced an *adaptive compute-phase prediction* scheme. Adaptive compute-phase prediction allows us to determine the distance threshold on the fly by monitoring memory access characteristics of a thread. The distance threshold determined adaptively tolerates short distortions and bursts that can be seen in compute phase, as it is in the original work. More importantly, adaptively determined distance threshold helps to predict the execution phase changes earlier compared to predefined distance threshold. The illustration of original compute-phase prediction is given in Figure 1. Similarly, the illustration of adaptive compute-phase prediction is given in Figure 2.

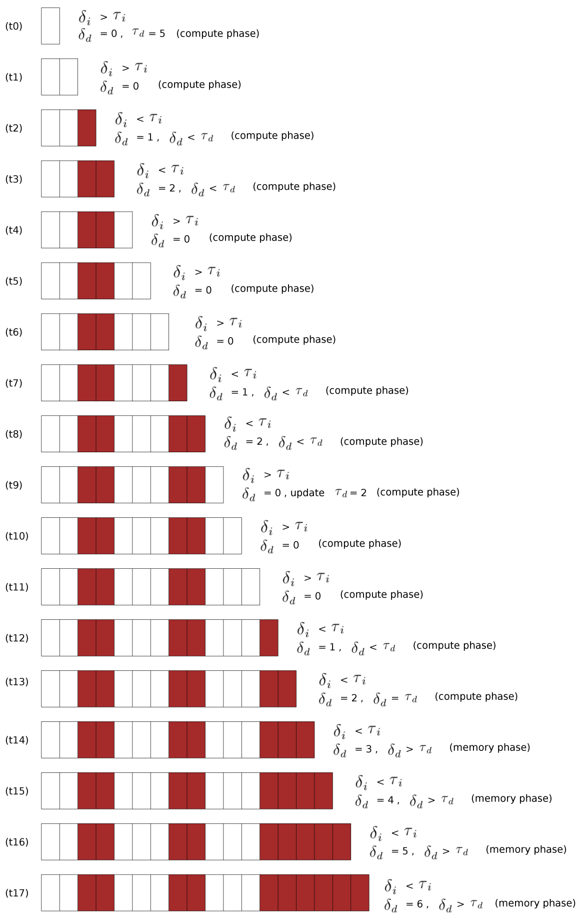


(t0) $\delta_i > \tau_i$ $\delta_d = 0$, $\tau_d = 5$ (compute phase)
(t1) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t2) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t3) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d < \tau_d$ (compute phase)
(t4) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t5) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t6) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t7) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t8) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d < \tau_d$ (compute phase)
(t9) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t10) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t11) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t12) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t13) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d < \tau_d$ (compute phase)
(t14) $\delta_i < \tau_i$ $\delta_d = 3$, $\delta_d < \tau_d$ (compute phase)
(t15) $\delta_i < \tau_i$ $\delta_d = 4$, $\delta_d < \tau_d$ (compute phase)
(t16) $\delta_i < \tau_i$ $\delta_d = 5$, $\delta_d = \tau_d$ (compute phase)
(t17) $\delta_i < \tau_i$ $\delta_d = 6$, $\delta_d > \tau_d$ (memory phase)

Figure 1: Default compute-phase prediction.

In Figure 1, the predefined distance threshold (τ_d) is set to five. Light boxes indicate that interval counter constraint is satisfied (i.e δ_i exceeds τ_i). Dark boxes indicate that interval counter constraint is not satisfied (i.e δ_i stays below τ_i). The distance counter (δ_d) is incremented if interval counter constraint is not satisfied and reset otherwise. When the distance counter (δ_d) exceeds the distance threshold (τ_d is five in this illustration), the thread is considered to be in memory phase. The thread is considered to be in compute phase when it satisfies interval counter constraint again.

On the other hand, in Figure 2, our adaptive compute phase prediction scheme observes that the interval counter constraint is satisfied, except consecutive two cache misses. By using this observation, our adaptive compute phase prediction determines that there is no need to consider a thread in compute phase if the distance counter (δ_d) exceeds two. Whenever the third consecutive access that does not satisfy interval counter constraint is occurred, adaptive compute-phase prediction concludes that a thread exits compute phase and goes into memory phase. Note that, adaptive compute-phase prediction can detect the change in execution phase much earlier. Thus, adaptive compute-phase prediction increases the accuracy of prediction and reduces the time required which leads to improved overall performance and fairness.



(t0) $\delta_i > \tau_i$ $\delta_d = 0$, $\tau_d = 5$ (compute phase)
(t1) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t2) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t3) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d < \tau_d$ (compute phase)
(t4) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t5) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t6) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t7) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t8) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d < \tau_d$ (compute phase)
(t9) $\delta_i > \tau_i$ $\delta_d = 0$, update $\tau_d = 2$ (compute phase)
(t10) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t11) $\delta_i > \tau_i$ $\delta_d = 0$ (compute phase)
(t12) $\delta_i < \tau_i$ $\delta_d = 1$, $\delta_d < \tau_d$ (compute phase)
(t13) $\delta_i < \tau_i$ $\delta_d = 2$, $\delta_d = \tau_d$ (compute phase)
(t14) $\delta_i < \tau_i$ $\delta_d = 3$, $\delta_d > \tau_d$ (memory phase)
(t15) $\delta_i < \tau_i$ $\delta_d = 4$, $\delta_d > \tau_d$ (memory phase)
(t16) $\delta_i < \tau_i$ $\delta_d = 5$, $\delta_d > \tau_d$ (memory phase)
(t17) $\delta_i < \tau_i$ $\delta_d = 6$, $\delta_d > \tau_d$ (memory phase)

Figure 2: Adaptive compute-phase prediction.

5. ADAPTIVE THREAD PRIORITIZATION

As described earlier, threads are classified into two groups. Threads in compute phase are prioritized over threads in memory phase. However, it is possible to have multiple threads in compute phase. In the scheduler of Ishii et al., the memory requests of threads in compute phase are serviced in the order they have received. Although it allows threads to make progress and keep cores busy, it misses possible performance benefits that could be obtained through fine-grained prioritization among threads of the same group.

We observed that prioritizing threads based on their potentials of making more progress on their execution increases the system performance even further. For this reason, we enhanced prioritization scheme of Ishii et al. in a way that threads in the same group are prioritized based on their potentials of making progress in their execution. when their memory requests serviced by the memory controller. We call this fine-grained prioritization scheme as *adaptive thread prioritization* since the priorities of threads are determined on the fly.

The usage of adaptive thread prioritization differs for threads in different groups (i.e., memory-non-intensive and memory-intensive). Adaptive thread prioritization works for threads in compute phase as follows. Among the threads in compute phase, the one that has the highest potential to make more progress is prioritized. On the other hand, threads in memory phase are prioritized based on whether they exhibit page hit and rank/bank locality. The adaptive thread prioritization is used as a tie breaker for threads in memory phase when there are multiple threads that exhibit page hit or rank/bank locality with recent memory accesses.

The reason behind prioritizing threads in compute phase based on the progress they can make (i.e., employing adaptive thread prioritization) is to keep cores busy as much as possible. While cores are kept busy to execute threads in compute phase, the memory controller can service to memory requests of other threads. Thus, a thread that has more potential to keep a core busy for a longer period of time is prioritized over others.

On the other hand, if there is no thread in compute phase, then the main goal becomes to maximize memory throughput and reduce latency. For this reason, threads that exhibit row-buffer hit or rank/bank locality are given higher priorities. If there is no row-buffer hit, then the threads accessing the same bank/rank that was accessed recently are prioritized. When there are multiple threads that exhibit row-buffer hit, or bank/rank locality with recent memory access, then the adaptive thread prioritization is used to decide which thread is going to be prioritized.

We also employ aging in order to provide fair access to the memory. After a certain period of time, regardless of whether a thread is in compute phase or not, it is given the highest priority to avoid starvation. Threads that have low memory-level parallelism are also prioritized over other threads to let them finish their memory operations and continue on their execution as soon as possible.

6. CONCLUSION

We introduce an adaptive compute-phase prediction and thread prioritization algorithm for embedded memory scheduling. It adaptively decides whether a thread is in compute phase or in memory phase. The threads in compute phase are prioritized among the threads in memory phase. Also, the threads in compute phase are prioritized among themselves based on the potential progress they can make in their execution. Compared to FR-FCFS and CP-WO, it reduces the execution time of workloads up to 23.6% and 12.9%, respectively. Similarly, it reduces the total system power compared to FR-FCFS and CP-WO by 1% and 1.6%, respectively.

7. REFERENCES

[1] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, H., N. Udipi, Aniruddha, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the utah simulated memory module. Technical report, University of Utah, UUCS-12-002, 2012.

[2] I. Hur and C. Lin. A comprehensive approach to dram power management. In *High Performance Computer Architecture, 2008. IEEE 14th International Symposium on*, pages 305 –316, feb. 2008.

[3] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 39–50, Washington, DC, USA, 2008.

[4] Y. Ishii, K. Hosokawa, M. Inaba, and K. Hiraki. High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap. In *Proceedings of 3rd JILP Workshop on Computer Architecture Competitions*, Portland, OR, USA, 2012.

[5] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, Washington, DC, USA, 2010.

[6] MSC. 3rd jilp workshop on computer architecture competitions (jwac-3): Memory scheduling championship (msc). http://www.cs.utah.edu/ rajeev/jwac12/, jun. 2012.

[7] J. Mukundan and J. Martínez. Morse: Multi-objective reconfigurable self-optimizing memory scheduler. In *High Performance Computer Architecture, 2012. IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[8] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, Washington, DC, USA, 2007.

[9] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 63–74, Washington, DC, USA, 2008.

[10] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, New York, NY, USA, 2000.

[11] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 375–384, Washington, DC, USA, 2010.