An Analytical Model to Predict Performance Impact of DRAM Bank Partitioning

Dong Uk Kim
dongukim12@skku.edu

Seokju Yoon
sukju86@skku.edu

Jae W. Lee
jaewlee@skku.edu

Department of Semiconductor Systems Engineering
Sungkyunkwan University
Suwon, Korea

ABSTRACT

The main memory system is one of the most important shared resources in multicore platforms. To maximize bandwidth and minimize latency, modern DRAM systems exploit spatial locality in a memory access stream. However, multiple memory request streams from multiple concurrent threads are interleaved, hence killing the spatial locality of DRAM accesses. This increases bank conflicts and decreases row buffer hit rates. To address this problem, we advocate workload-aware, non-uniform DRAM bank partitioning to improve overall system throughput and fairness. To make best use of a limited number of DRAM banks offered by multicore platforms, we propose an analytical approximation model that predicts the program slowdown when the number of allocated banks is varying and apply it to find an optimal bank allocation for a given multi-programmed workload. Our preliminary evaluation shows promising results—the non-uniform bank partitioning guided by the prediction model improves overall system throughput by up to 3% (and 0.8% on average) and fairness by up to 26% (and 7.5% on average) over static uniform partitioning.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memoryies—*Dynamic memory (DRAM)*; D.4.2 [**Operating Systems**]: Storage Management—*Main memory*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Management, Performance, Design

Keywords

DRAM, Partitioning, Optimization, Memory management

1. INTRODUCTION

The main memory system will remain as one of the most important shared resources in multicore platforms in the foreseeable future. At odds with their name, modern DRAMs (Dynamic Random Access Memory) are not truly random access devices but are 3-D memory devices with dimensions of (bank, row, column). To maximize bandwidth and minimize latency, modern DRAM systems exploit both *bank parallelism* and *row buffer locality*. They service multiple outstanding memory requests in parallel if their addresses are mapped to different banks. Also, an entire memory row is copied over to the per-bank row buffer first before column accesses are performed. This lowers both access latency and power consumption since accessing the row buffer is much more efficient than accessing the entire memory bank.

Multicore platforms are called upon to execute multiple programs efficiently via space sharing. In this multi-programmed environment, multiple request streams from different threads are interleaved, hence killing the spatial locality of DRAM accesses [1]. This increases bank conflicts and decreases row buffer hit rates. Besides, a memory performance hog may slow down all the other concurrent threads by filling up memory request queues to increase queue occupancy.

Researchers have proposed a number of solutions to address this problem, most of which are based on either memory scheduling or bank partitioning [1, 2, 3, 4, 5, 6, 7, 8, 9]. In memory scheduling-based solutions, the memory controller schedules requests out of order to recover some of the lost spatial locality [2, 3, 4, 5, 6, 7]. Most scheduling algorithms prioritize requests to an open row to improve memory throughput while providing quality-of-service (QoS) to prevent starvation. However, bank interferences cannot be eliminated completely since all banks are still shared by all threads. Moreover, these solutions usually require hardware modifications in the memory controller, hence cannot be deployed in existing systems.

Alternatively, there are solutions based on memory bank partitioning to eliminate inter-process bank conflicts [1, 8, 9]. By partitioning DRAM banks and allocating partitions to different threads, one can break the performance coupling between concurrent threads caused by inter-thread DRAM contention at the cost of reduced bank parallelism. For example, Lei et al. proposes a page-coloring based bank-level partitioning mechanism, called BPM, which statically partitions DRAM banks into equal sizes and distributes them fairly to all available cores [9]. However, static allocation of DRAM banks often leads to suboptimal performance since threads may have different sensitivities to the number of

MSPC'13, June, 2013, Seattle, Washington.

available banks. To compensate for the reduced bank parallelism of each thread, Jeong et al. proposes to employ memory sub-ranking, which effectively increases the number of independent banks [1]. Unfortunately, support for memory sub-ranking requires hardware modifications to existing systems.

We advocate *workload-aware*, *non-uniform* DRAM bank partitioning to maximize overall system throughput while minimizing the maximum slowdown (unfairness). Our premise is that programs have different sensitivities to the number of banks. To make best use of a limited number of DRAM banks without extra hardware support, it is necessary to support non-uniform bank allocation to concurrent threads contending to access main memory. When allocating banks to a thread, it is highly desirable to take into account the thread's performance slowdown curve with varying number of DRAM banks allocated.

Therefore, this paper proposes an analytical model that predicts the program slowdown when the number of allocated banks is varying. The input to this model is a *circular sequence profile* [10] of the program without bank partitioning. This circular sequence profile captures the program's temporal reuse behaviors on row buffers and can be easily obtained online or offline. The output is the program slowdown as a function of the number of allocated banks. This model not only reduces the burden of performance characterization over a myriad of programs, platforms, and bank partitions, but also gives an insight into what types of programs are susceptible to performance degradation with fewer DRAM banks.

Using this model, we find an optimal bank allocation for a given multi-programmed workload, and our preliminary evaluation with 25 multi-programmed workloads shows promising results—the non-uniform bank partitioning guided by the prediction model improves overall system throughput by up to 3% (and 0.8% on average) and fairness by up to 26% (and 7.5% on average) over static uniform partitioning. Compared to the baseline system with no bank partitioning, it improves overall system throughput by up to 8% (and 2% on average) and fairness by up to 19% (and 4.5% on average).

2. MOTIVATION

2.1 Optimal DRAM Bank Allocation

Programs have different degrees of bank parallelism, spatial locality, and memory latency tolerance, which determine their performance sensitivity to the number of allocated banks. To illustrate this, Figure 1 shows the IPCs of 21 benchmarks with varying number of DRAM banks from 2 to 64, normalized to the IPC with 64 banks. For namd, for example, one bank suffices to achieve over 95% of the IPC with 64 banks. However, tigr needs 32 banks to achieve the same level of performance. The performance sensitivities to the number of DRAM banks vary widely across the benchmarks in Figure 1.

Therefore, workload-aware, asymmetric DRAM bank allocation is necessary to maximize overall system throughput while providing quality-of-service (QoS). Figure 2 shows such an example with two programs, tigr (from Biobench [11])

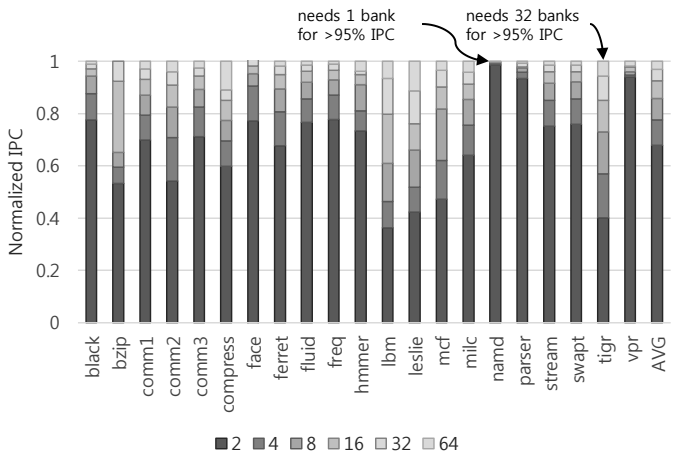

Figure 1: IPC with varying number of DRAM banks (normalized to the 64-bank case).

and blackscholes (from Parsec benchmark suite [12]), assuming 16 DRAM banks are available per core as in [9]. While blackscholes performs well with a relatively small number of banks, the performance of tigr degrades much faster with decreasing number of banks. In this case, by unevenly allocating 24 and 8 banks to tigr and blackscholes, respectively, the overall system throughput is improved by 2% and the performance-fairness product (PFP) [13] by 8%, compared to the uniform partitioning.

An increase in average memory access latency with fewer banks is affected by two factors. First, using fewer banks reduces row hit rate since there is a higher chance for intervening requests to fall between a pair of adjacent requests to the same row, which would not have disturbed the pair with more banks. This increases the time to access a DRAM device (called *DRAM access time*) for precharging and activation operations. Second, with fewer banks, a bank must service more requests on average, which in turn increases the waiting time of a request in the request queue (called *queueing delay*). Note that, overall memory latency can be represented by a sum of DRAM access time and queueing delay.

The two components of memory latency are affected to different extents when fewer banks are allocated. Figure 3 shows an increase in DRAM access time and queueing delay with varying number of banks for bzip and mcf. In bzip both DRAM access time and queueing delay increase rapidly with fewer than 16 banks. In contrast, only the queueing delay increases rapidly for mcf, whereas DRAM access time remains almost constant. To predict the average memory latency of a program when the number of banks allocated to it varies, both components should be carefully analyzed and modeled.

In addition to memory latency, a program's memory-level parallelism and latency tolerance also affect the slowdown of the program when fewer banks are allocated. Figure 4 shows an increase in memory (read) latency for each program and its slowdown when the number of banks is reduced from 64 to 2. All programs are divided into three groups and sorted in an ascending order of memory latency increase. In the

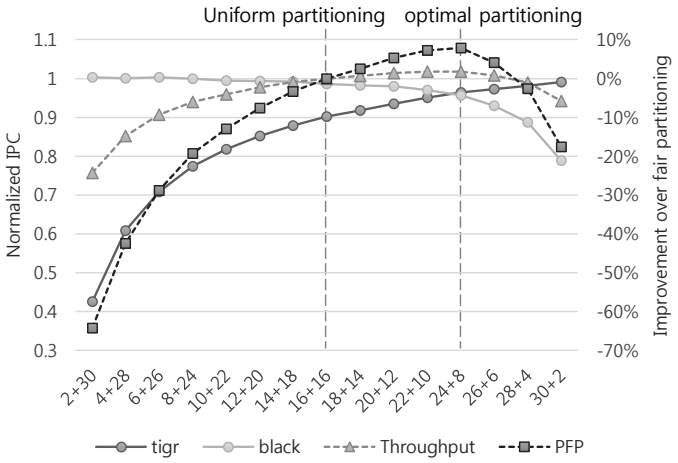

Figure 2: Optimal DRAM bank partitioning for two programs running concurrently on 32 banks. 2+30 on X axis indicates allocation of 2 banks to tigr and 30 banks to blackscholes. With an asymmetric allocation of 24+8, overall throughput improves by 2% and performance-fairness product (PFP) by 8%, compared to static fair allocation (16+16).

high misses-per-kilo-instructions (MPKI) group, the amount of increase in memory latency is a very good predictor of program slowdown. However, in the low MPKI group, even if the memory latency increases by over 50%, its impact on program slowdown is limited. From these observations, we find that a product of memory latency increase and MPKI serves as a good predictor of program slowdown when fewer banks are allocated to the program.

2.2 Physical Address Mapping for Software-Only Bank Partitioning

There are several recent proposals for software-based DRAM bank partitioning with or without hardware support [1, 9, 14]. They modify the memory frame allocator to partition DRAM banks into multiple sets (or colors) and map memory frames used by different cores to different sets, hence eliminating row buffer interferences. Figure 5 shows an example of virtual page to physical frame to DRAM address mapping assuming 2 channels, 4 ranks and 4 banks. In this setup, we can create 16 sets that can be flexibly distributed to multiple concurrent threads purely in software. This can be done by using 2 rank bits and 2 bank bits as a set ID when allocating a physical frame. Note that the channel ID bit falls on the page offset field, which cannot be directly controlled by the memory allocator, and cannot be used to create additional sets.

One challenge in implementing a software-only DRAM bank partitioning system is to find out the DRAM address bit mapping in commodity systems. Lei et al. [9] and Park et al. [15] come up with similar ideas to address this challenge. They generate requests to access different physical frames and measure memory latency to cluster long and short latency groups. By exploiting the knowledge of bank parallelism and locality behavior, they effectively extract the DRAM mapping information. We build on their framework that enables software-only DRAM bank partitioning on commodity systems. The remaining question is how to allocate sets to concurrent threads to maximize overall throughput and provide QoS. The rest of this paper presents (1) a performance model that predicts a program's slowdown when the number of allocated banks is reduced; and (2) its application to find an optimal partitioning configuration for a given multi-programmed workload.

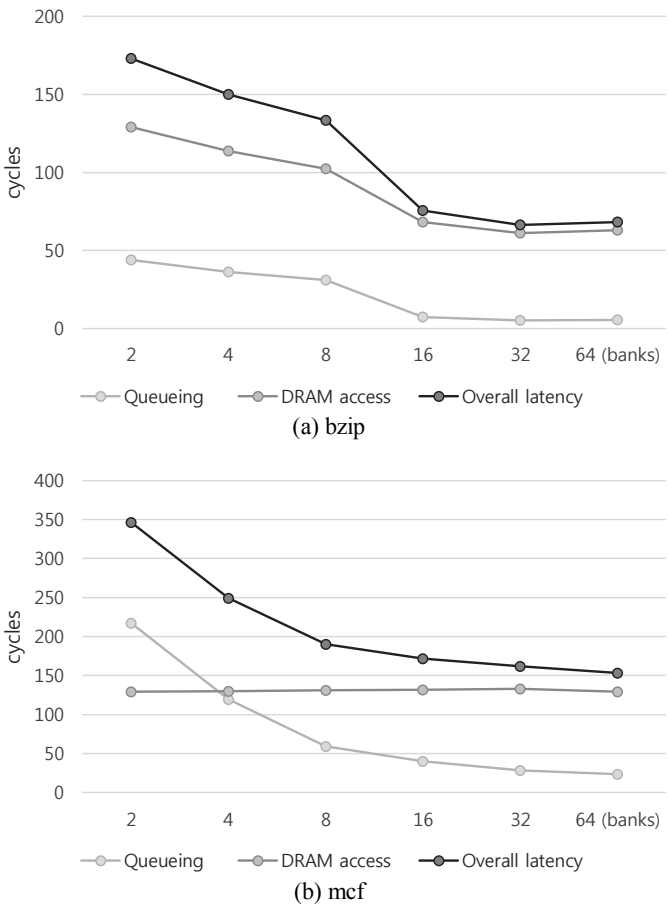

Figure 3: Increase in DRAM access time and queueing delay with varying number of banks for (a) bzip and (b) mcf.

3. PERFORMANCE PREDICTION MODEL

To find an optimal configuration of DRAM bank allocation, it is necessary to characterize the performance curve of each program in a multi-programmed workload with varying number of banks as in Figure 1. For this, an analytical model that *predicts* the program's slowdown when using only a subset of DRAM banks would be of great value. This model not only reduces the burden of performance characterization over a myriad of programs, platforms, and bank partitions, but also gives an insight into what types of programs are susceptible to performance degradation with fewer DRAM banks.

We present a prediction model that estimates the program slowdown caused by bank partitioning, compared to standalone execution without partitioning. The input to this

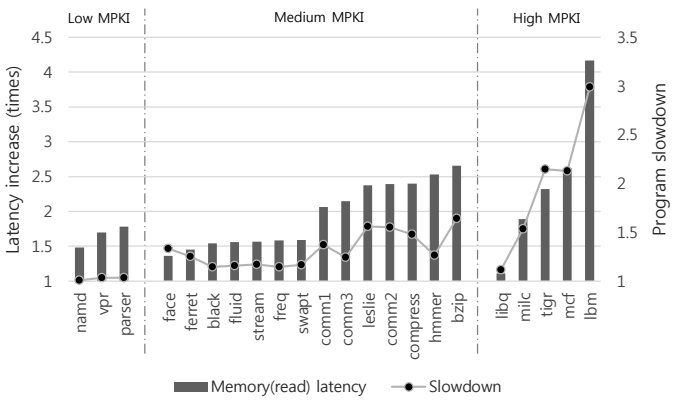

Figure 4: Correlation between increase in memory (read) latency and program slowdown

...	21	20	19	18	17	16	15	14	13	12	...	7	6	5	4	3	2	1	0
Virtual Page Number										Page Offset									
Memory allocator (software)																			
Physical Frame Number										Page Offset									
DRAM address mapping (hardware)																			
Row						Rank	Bank	Column					Ch	Column					

Figure 5: Virtual/physical to DRAM address mapping (modified from a figure in [1]).

model is a *circular sequence profile* [10] of the program without bank partitioning. This circular sequence profile captures the program's temporal reuse behaviors on row buffers and can be easily obtained online or offline [10]. Assuming B_{MAX} denotes the number of banks available in the system, this model outputs the ratio of IPC when using only B_{target} banks (where $B_{target} \leq B_{MAX}$) to IPC when using all B_{MAX} banks. Extending the definition in [10], we define a circular sequence as follows:

Definition. The circular sequence of DRAM read accesses that is terminated by the i-th memory read, denoted by $CSEQ_i$, is a series of n_i DRAM read accesses to d_i distinct DRAM banks, where the first and last accesses fall on the same bank, and there are no other accesses to that bank.

Figure 6 illustrates three circular sequences, each of which is characterized by three parameters: d_i, n_i, and m_i. d_i and n_i denote how many distinct banks are accessed, and how many read requests are there within $CSEQ_i$, respectively. m_i is *hit* if the first and last accesses in $CSEQ_i$ fall on the same row; otherwise, it is *miss*. Note that, the number within each box denotes its read sequence number.

The key insight of our proposal is that the statistical properties of the three random variables (d_i, n_i, and m_i) reveal a program's performance sensitivity to the number of allocated DRAM banks. For example, a program with high n_i generally requires more banks for good performance. d_i gives us hints in estimating queueing delay. Using m_i, we can estimate row hit rates. We only consider read requests in our model since the performance impact of write requests is secondary.

The rest of this section is organized as follows. Section 3.1

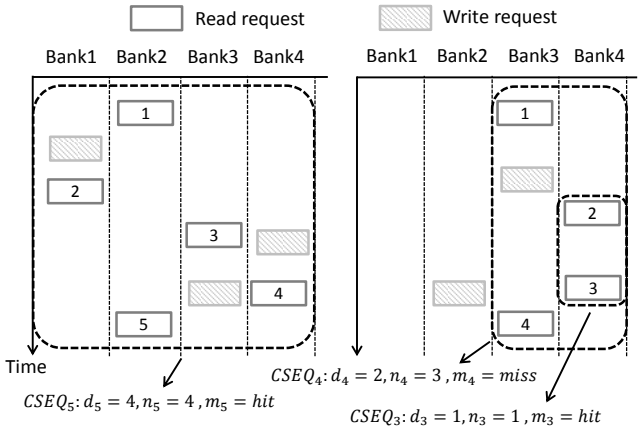

Figure 6: Circular sequence for i-th read request ($CSEQ_i$) characterized by three parameters: d_i (distinct banks accessed), n_i (number of read requests), and m_i (row hit or miss).

and Section 4.2 describe prediction models for the two delay components constituting memory latency, DRAM access time and queueing delay, respectively. Exploiting these two models, Section 3.3 presents how to estimate the program slowdown caused by bank partitioning as a function of B_{target} and how to find an optimal configuration of bank partitions for a given multi-programmed workload.

3.1 Row Hit Rate Prediction

DRAM access time can be approximated as a function of row hit rate since it is a weighted sum of row miss access time (tRP+tRCD+tCL+BL/2*tCK) [1] and row hit access time (tCL+BL/2*tCK). Therefore, we focus on a prediction model for row hit rate when B_{target} banks are used, denoted by $HitRate(B_{target})$.

Figure 7 illustrates an example of bank partition when 4 banks are allocated to the program (i.e., B_{target}=4). Assuming Banks 1-4 are used, those requests originally mapped to outside these banks will have to relocated into the allocated banks. Since the expected value of d_i, denoted by $E(d_i)$, indicates how many distinct banks are accessed on average by a circular sequence, the expected number of banks accessed outside the allocated banks (hence those that should be relocated) is given as follows:

$$AvgRelocatedBanks(ARB) = (E(d_i)-1) \times \frac{B_{MAX} - B_{target}}{B_{MAX} - 1}$$

Request relocation can turn a row hit into a row miss if a relocated request from outside of the allocated banks intervenes between a pair of requests yielding a row hit. Using the same example in Figure 7, there are three possible relocation scenarios as shown in Figure 8. If either Bank 7 or Bank 8 is relocated onto Bank 2, it will kill the row hit of the second request in Bank 2 (labeled 9). If both Bank 7 and 8 are relocated to banks other than Bank 2 (as in Figure 8), the original row hit will be preserved. To generalize

[1]These DRAM timing parameters mean: tRP (precharge command period), tRCD (activate to read delay), tCL (CAS latency), BL (burst length), and tCK (clock period).

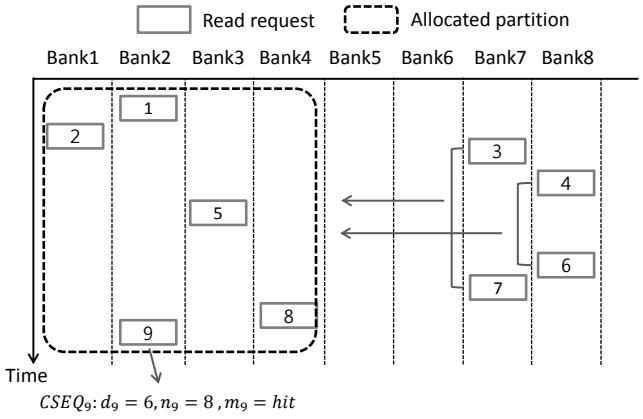

Figure 7: Example of extra row miss caused by request relocation

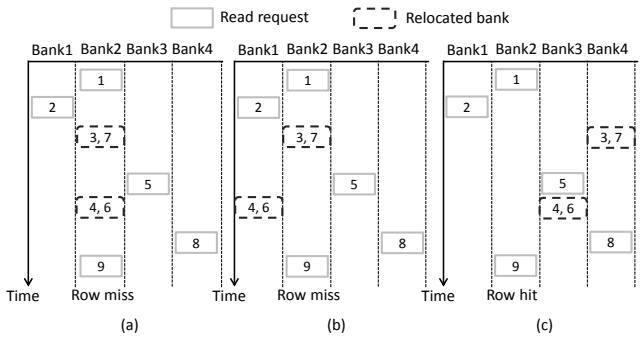

Figure 8: Three cases of bank relocation

it, the probability of preserving an original row hit after the relocation is given as follows:

$$RowHitPreservProb(RHP) = (\frac{B_{target}-1}{B_{target}})^{ARB}$$

Note that, a circular sequence with d_i=1 always yields a row hit since there are no intervening requests between the two requests constituting the circular sequence. Hence, row hit requests can be divided into two sets of d_i=1 and $d_i \geq 2$. Then the row hit rate can be represented as a sum of the two fractions contributed by the two sets, respectively. $HitRate_{di=1}$ denotes the first fraction, and $HitRate_{di\geq 2}$ the second fraction. $HitRate_{di=1}$ is not affected by bank relocation behaviors, but $HitRate_{di\geq 2}$ is. Therefore, the row hit rate with a partition of B_{target} banks is given as follows:

$$HitRate(B_{target}) = HitRate_{di=1}(B_{MAX}) + HitRate_{di\geq 2}(B_{MAX}) \times RHP$$

3.2 Queueing Delay Prediction

The second delay component that constitutes memory latency is queueing delay. As Figure 3 shows, queueing delay can increase rapidly if the number of allocated banks decreases below a certain threshold (which is 16 in this example).

Our prediction model uses d_i to estimate an increase in queueing delay with several modifications. First, requests outside the allocated banks are relocated into the allocated banks, which increases queueing delay. Therefore, the *bank shortage* of a circular sequence, defined as the difference between d_i and B_{target}, is a good estimate of how many extra requests are placed into the B_{target} banks. Second, as bank utilization becomes low, queueing delay converges to zero. Hence, our model considers only those circular sequences with $d_i > B_{target}$. Third, high n_i means there is a large interval in accessing the same bank by the first and last requests in a circular sequence, which in turn implies low queueing delay. So, we assume $CSEQ_i$ does not increase queueing delay if $n_i > MPKI$. Finally, the queueing delay is affected also by an increased DRAM access time, and we take it into account in calculating an increase in the queueing delay (denoted by $\Delta QDelay$).

We first define bank shortage (BS_i) and bank shortage flag (BSF_i) for the i-th circular sequence as follows:

$$BS_i = \begin{cases} d_i - B_{target} & \text{for} \quad d_i > B_{target} \;\&\; n_i < MPKI \\ 0 & Otherwise \end{cases}$$

$$BSF_i = \begin{cases} 1 & \text{for} \quad d_i > B_{target} \;\&\; n_i < MPKI \\ 0 & Otherwise \end{cases}$$

Then the increase in queueing delay can be represented as follows:

$$\Delta QDelay(B_{target}) = QDelay(B_{target}) - QDelay(B_{MAX})$$
$$= E(BSF_i) \times \frac{E(BS_i)}{B_{target}} \times RowMissAccessTime +$$
$$(DramAccessTime(B_{target}) - DramAccessTime(B_{MAX}))$$

3.3 Finding Optimal Bank Allocation

Exploiting the two prediction models for $HitRate(B_{target})$ and $\Delta QDelay(B_{target})$, we can calculate an estimated slowdown of the program when the bank count decreases from B_{MAX} to B_{target} as follows:

$$Slowdown(B_{target}) = \frac{MPKI \times ReadLatency(B_{target}) + 1000 \times CPI}{MPKI \times ReadLatency(B_{MAX}) + 1000 \times CPI}$$

The first term in both numerator and denominator represents average memory read cycles in executing 1000 instructions; the second term average CPU cycles. MPKI and CPI can be easily measured by using performance counters modern architectures provide or by simulation. In this work, we use the CPI without partitioning (i.e., with B_{MAX} banks).

Obtaining *ReadLatency* is trickier. Among the two delay components, DRAM access time can be readily calculated as a function of row hit rate, and DRAM timing parameters as discussed in Section 3.1. To estimate $QDelay(B_{target})$, which is a sum of $QDelay(B_{MAX})$ and $\Delta QDelay(B_{target})$ as given in Section 4.2, we should estimate $QDelay(B_{MAX})$ first. Based on our characterization of various programs, we create a prediction model in Table 1 with 6 intervals considering MPKI and *DelayFactor*, which is defined as follows:

$$DelayFactor = \frac{1}{E(n_i) \times E(d_i)}$$

MPKI \ Delay factor	DF < 0.05	$0.05 \leq$ DF < 0.08	$0.08 \leq$ DF
Low (< 1)	0	0	$RowMissAccessTime/QDF$
High (> 1)	0	$RowMissAccessTime/QDF$	$RowMissAccessTime$

Table 1: Prediction table for $QDelay(B_{MAX})$

Queueing delay degradation factor (QDF) in Table 1 determines how fast the queueing delay decreases as either MPKI or delay factor decreases. This is a platform-specific parameter, and a range of 2-3 works well in our setup. This prediction model $QDelay(B_{MAX})$ has limited accuracy due to many simplifications. However, it faithfully captures the shape of a slowdown curve to help make a decision of bank partitioning.

Using the slowdown model, we find an optimal bank allocation to the programs running concurrently. The optimization goal is to equalize the slowdowns of all the programs to improve fairness as well as overall throughput. To achieve this, we start from uniform, fair allocation of memory banks. Then we cluster programs into multiple groups based on their normalized IPCs. Groups with high IPCs become donor groups, and groups with low IPCs become recipient groups. We enforce more than a half of all programs to belong to donor groups. In transferring banks, the minimum unit of transfer is two since a channel bit typically falls in a page offset field as shown in Figure 5. We also enforce minimum and maximum number of banks allocated to a program—a minimum of a half of the fair share and a maximum of twice the fair share. This rule of upper and lower bounds does not apply to programs with low MPKIs since these programs are mostly insensitive to memory latency.

4. EVALUATION

We evaluate the performance and fairness of DRAM bank partitioning guided our prediction models using USIMM [13], a trace-driven DRAM timing model simulator. The baseline DRAM configuration is a 4GB DDR3 system with 2 channels, 4 ranks per channel, and 8 banks per rank. There are 32 allocation units (corresponding to 64 banks) since the channel bit falls on page offset to which we do not have a direct control. We plan to implement our proposed model and evaluate it in a real system in the future. For evaluation we use 25 multi-programmed workloads taken from SPEC CPU2006 [16], PARSEC [12], Biobench [11], and Intel commercial workloads [13] as summarized in Table 2 and Table 3.

To measure system throughput and fairness, we use *Weighted Speedup* (WS) [5] and *Maximum Slowdown* (MS) [5], respectively. *Performance Fairness Product* (PFP) [5] summarizes the overall system improvements. We compare our proposed model to the baseline memory system without partitioning and uniform bank partitioning proposed in [9].

$$Weighted\ Speedup(WS) = \sum \frac{IPC_{shared}}{IPC_{alone}}$$

Label	Name	Benchmark suite	MPKI	RB Hit Rate
black	blackscholes	PARSEC [12]	4.57	62%
face	facesim		10.37	85%
fluid	fluidanimate		4.72	63%
freq	freqmine		4.42	63%
stream	streamcluster		5.57	63%
swap	swaptions		5.15	62%
tigr	tigr	Biobench [11]	28.86	18%
hmmer	456.hmmer		3.5	89%
compress	129.compress	SPEC CPU 95 [17]	6.91	60%
bzip	256.bzip2	SPEC CPU 2000 [18]	3.93	95%
vpr	175.vpr		0.39	76%
parser	197.parser		0.7	84%
lbm	470.lbm	SPEC CPU 2006 [16]	38.35	87%
leslie	leslie3d		9.45	85%
libq	libquantum		20.2	81%
mcf	429.mcf		19.94	21%
milc	433.milc		14.49	89%
namd	444.namd		0.56	97%
comm1	comm1	Commercial workloads from Intel [13]	9.06	64%
comm2	comm2		13.25	43%
comm3	comm3		5.12	51%

Table 2: Benchmarks for evaluation with MPKI (last level-cache misses per 1000 instructions) and RB Hit Rate (row-buffer hit rate).

$$Maximum\ Slowdown(MS) = \max \frac{IPC_{alone}}{IPC_{shared}}$$

$$Performance\ Fairness\ Product(PFP) = \frac{WS}{MS}$$

4.1 System Throughput and Fairness

Figure 9 shows weighted speedups and maximum slowdown normalized to the numbers without bank partitioning. Overall, non-uniform bank partitioning guided by the prediction model in Section 3 improves weighted speedups up by up to 8% with an average of 2% over the baseline. Although uniform bank partitioning outperforms non-uniform partitioning in a few cases such as Case 19 and Case 22, the latter generally performs better than the former.

For fairness non-uniform bank partition shows much better fairness than uniform bank partition. In 8-programmed workloads, only 4 allocation units (i.e., 8 banks) are allocated to each program, and some programs suffer from

No.	Benchmarks (in the order of MPKI)	No.	Benchmarks (in the order of MPKI order)
1	leslie, fluid, black, freq, bzip, parser, namd, vpr	2	comm2, comm1, stream, swapt, comm3, fluid, namd, vpr
3	milc, comm2, leslie, fluid, black, bzip, hmmer, namd	4	mcf, leslie, comm1, fluid, freq, hmmer, parser, namd
5	milc, ferret, face, compress, stream, swapt, hmmer, vpr	6	libq, comm2, comm1, comm3, fluid, black, freq, parser
7	lbm, ferret, stream, swapt, bzip, hmmer, parser, namd	8	ferret, comm1, stream, fluid, black, freq, hmmer, namd
9	tigr, libq, milc, comm1, stream, swapt, fluid, hmmer	10	ferret, compress, stream, swapt, comm3, freq, hmmer, parser
11	lbm, leslie, compress, black, bzip, parser, namd, vpr	12	face, comm3, fluid, black, freq, bzip, parser, namd
13	libq, ferret, leslie, comm3, fluid, black, freq, vpr	14	libq, comm2, face, comm1, stream, black, parser, namd
15	ferret, comm1, stream, swapt, fluid, black, hmmer, vpr	16	tigr, leslie, compress, bzip
17	compress, fluid, freq, bzip	18	comm2, ferret, leslie, stream
19	lbm, libq, compress, comm3	20	lbm, leslie, black, namd
21	lbm, tigr, namd, vpr	22	libq, ferret, fluid, bzip
23	face, leslie, black, parser	24	milc, comm2, stream, swapt
25	libq, face, freq, hmmer		

Table 3: Multi-programmed workloads

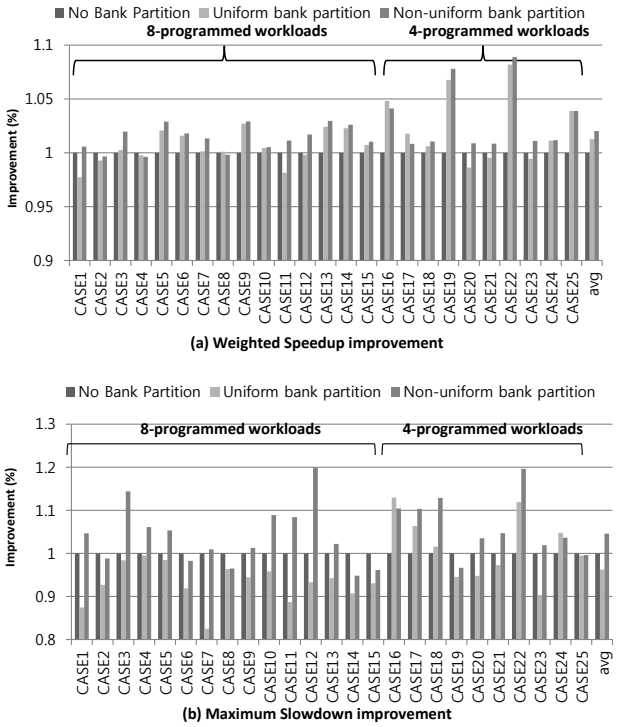

Figure 9: Weighted Speedup (WS) and Maximum Slowdown (MS) improvements over baseline system with no partitioning

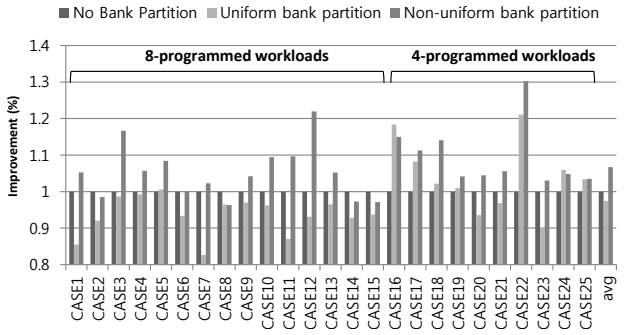

Figure 10: Performance Fairness Product (PFP) improvements over baseline system with no partitioning

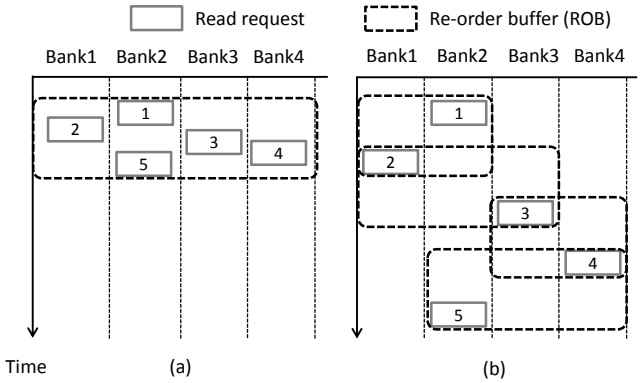

Figure 12: Prediction inaccuracy from phase behavior of request streams

significant performance degradation. This makes the fairness problem worse, but the non-uniform bank allocation improves performance and fairness significantly. For this reason, our model shows improvements by up to 19% with an average of 4.5% than the no-partitioning baseline and 8% than uniform bank partition.

Figure 10 shows PFP results. Our model improves PFP by an average of 6.7% over the baseline. It should be noted that several workloads (i.e, Case 2, Case 8, Case 14 and Case 15) yield lower PFPs than the baseline. These workloads contain comm1, whose slowdown is not accurately predicted by our model. This misprediction contributes to the degraded MS and PFP results. Uniform partitioning performs poorer than the baseline since it has high MS numbers by not considering each program's performance sensitivity to the number of allocated banks. In contrast, our work allocates banks non-uniformly by assigning more banks to bank-sensitive programs.

4.2 Prediction Accuracy

Figure 11 shows read latencies and slowdowns, both predicted and measured, for four benchmarks. Most of the prediction curves faithfully follow the real curves to help reason about the program performance. Especially, the error is very low when the bank count is over 8. Overall, an increase in read latency is a good indicator for program slowdown except for vpr, which has very low MPKI.

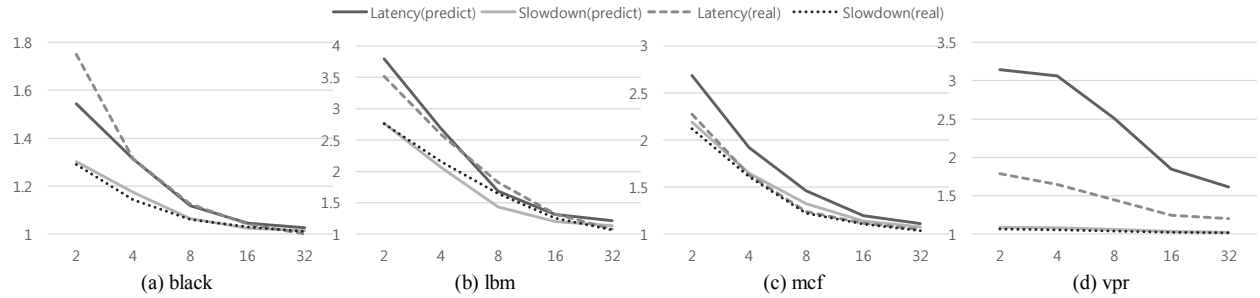

Figure 11: Application slowdown results by prediction and simulation

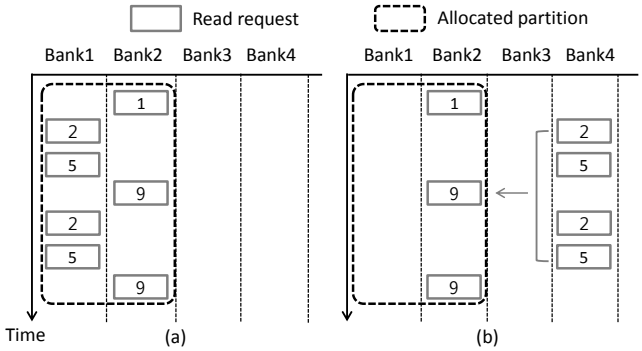

Figure 13: Example of extra memory latency caused by bank selection

vpr shows a relatively large error in estimating read latency, and the queueing delay component accounts for most of the error. The queueing delay is influenced by the burstiness of the request stream, but the queueing delay model in Section { does not take it into account. Even if two request streams have comparable long-term MPKIs, their temporal behaviors may differ widely. For example, a bursty request stream (as in Figure 12(a)) is likely to experience higher queueing delay than a non-bursty stream (as in Figure 12(b)). Note that the dotted rectangles capture those requests that occupy the re-order buffer at the same time.

We only take into account end-to-end statistical characteristics of a request stream without considering its temporal behavior, and this may lead to an inaccuracy in estimating overall slowdown. For example, Figure 13 shows two memory traces, which have the same d_i, n_i, m_i and MPKI. Theoretically, a bank relocation may not cause extra slowdown at all in a case like Figure 13, where relocated requests do not cause an extra row miss or an increase in queueing delay. However, our finding is that this kind of pathological cases are rare and that our model exhibits a good enough accuracy for a wide range of workloads to help make bank allocation decisions.

5. RELATED WORK

Memory scheduling: Memory controllers schedule memory requests according to each program's memory access behaviors [2, 3, 4, 5, 19]. ATLAS [4] and TCM [5] identify latency-sensitive applications using MPKI, and prioritize specific memory requests or applications for system throughput and fairness. These algorithms may relieve bank interferences among programs running concurrently, but do not completely eliminate them [1].

DRAM partition: Memory bank and channel partitioning lessen memory contention at a channel level [8] or at a bank level [1, 9]. Although these techniques often provide higher throughput, some of them may degrade fairness [1, 9]. Liu et al. [9] uniformly partition DRAM banks statically, which may yield suboptimal performance.

Memory related slowdown estimation: STFM [6] and MISE [7] estimate each application's slowdown rate to adapt the DRAM scheduling policy. STFM estimates the slowdown of an application by calculating the stall time caused by memory contention. To estimate the memory related slowdown, MISE executes an application in two different priority modes and compares their performance. STFM and MISE predict system performance at run-time to adapt to the dynamic system behaviors. However, they require hardware modifications. Since our work is based on bank partitioning without memory contention, it can simplify the memory access scheduler and provide better isolation.

6. CONCLUSION

We advocate software-controlled, workload-aware DRAM bank partitioning to improve overall system throughput and minimize maximum slowdown of a program. To make best use of a limited number of DRAM banks offered by multicore platforms, we propose an analytical model that predicts the program slowdown when the number of allocated banks is varying and apply it to find an optimal bank allocation for a given multi-programmed workload. The non-uniform bank allocation suggested by the model improves the overall throughput by up to 3% (and 0.8% on average) and fairness by up to 26% (and 7.5% on average) over static uniform partitioning.

We plan to integrate the prediction model with an existing software-only DRAM bank partitioning framework on a commodity platform. User-level APIs and runtime support are being designed to allow users to control bank allocations based on a program's needs. We also plan to investigate partitioning of DRAM banks and shared caches together to achieve maximum synergies between the two.

Acknowledgements

We would like to thank the members of Parallel Architecture and Programming Laboratory (PAPL) at Sungkyunkwan University for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the IT R&D program of MKE/KEIT [KI001810041244, Smart TV 2.0 Software Platform] and a research grant from Samsung Electronics.

7. REFERENCES

[1] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Michael Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *HPCA*, 2012.

[2] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Patt Yale N. Prefetch-aware shared-resource management for multi-core systems. In *ISCA*, 2011.

[3] Eiman Ebrahimi, Rustam Miftakhutdinow, and Chris Fallin. Parallel application memory scheduling. In *MICRO*, 2011.

[4] Yoongu Kim, Onur Mutlu Dongsu Han, and Mor Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[5] Yoongu Kim, Michael Papamichael, Onur Mutulu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[6] Onur Mutlu and Thomans Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.

[7] Lavanya Subramanian, vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutulu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.

[8] Sai Prashanth Muralidhara, Lavanya Subramanian, and Onur Mutlu. Reducing memory interfernece in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.

[9] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012.

[10] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inther-thread cache contetntion on a chip multi-processor architecture. In *HPCA*, 2005.

[11] K. Albayraktaroglu, A. Jaleel, M. Franklin X. Wu, B. Jacob, C.-W. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[13] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. USIMM: the utah simulated memory module. In *UUCS-12-002*, 2012.

[14] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In *IFIP International Conference on Network and Parallel Computing (NPC)*, 2010.

[15] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *ASPLOS*, 2013.

[16] Standard performance evaluation corporation. `http://www.spec.org/cpu2006`.

[17] Standard performance evaluation corporation. `http://www.spec.org/cpu95`.

[18] Standard performance evaluation corporation. `http://www.spec.org/cpu2000`.

[19] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.