DraMon: Predicting Memory Bandwidth Usage of Multi-threaded Programs with High Accuracy and Low Overhead

Wei Wang, Tanima Dey, Jack W. Davidson, and Mary Lou Soffa
Department of Computer Science
University of Virginia
{wwang, td8h, jwd, soffa}@virginia.edu

Abstract

Memory bandwidth severely limits the scalability and performance of today's multi-core systems. Because of this limitation, many studies that focused on improving multi-core scalability rely on bandwidth usage predictions to achieve the best results. However, existing bandwidth prediction models have low accuracy, causing these studies to have inaccurate conclusions or perform sub-optimally. Most of these models make predictions based on the bandwidth usage samples of a few trial runs. Many factors that affect bandwidth usage and the complex DRAM operations are overlooked.

This paper presents DraMon, a model that predicts bandwidth usages for multi-threaded programs with low overhead. It achieves high accuracy through highly accurate predictions of DRAM contention and DRAM concurrency, as well as by considering a wide range of hardware and software factors that impact bandwidth usage. We implemented two versions of DraMon: DraMon-T, a memory-trace based model, and DraMon-R, a run-time model which uses hardware performance counters. When evaluated on a real machine with memory-intensive benchmarks, DraMon-T has average accuracies of 99.17% and 94.70% for DRAM contention predictions and bandwidth predictions, respectively. DraMon-R has average accuracies of 98.55% and 93.37% for DRAM contention and bandwidth predictions respectively, with only 0.50% overhead on average.

1. Introduction

Although multi-core systems have become ubiquitous, various scalability issues severely limit the performance of these powerful platforms. Consequently, there is much ongoing research that investigates potential solutions to multi-core scalability challenges. For example, some studies have investigated core allocations for multi-threaded programs to achieve optimal scalable performance and avoid resource over-commitment [20, 35]. The performance limits of future multi-core processors and the design options for CPU cores have been explored to achieve better scalability [8, 29]. Methods to mitigate memory resource contention to improve scalability have also been developed [43]. In addition, there is research to build tools to help developers analyze and resolve scalability bottlenecks of existing programs [10, 12, 19].

Because memory bandwidth is one of the primary limits of multi-core systems scalability, these studies depend on bandwidth predictions to achieve the best results. More specifically, they all require a model to predict the memory bandwidth usage of a multi-threaded program when it executes with a certain number of cores/threads. For example, research on optimal core allocation required memory bandwidth prediction to avoid allocating more cores when the bandwidth is already saturated [20, 35].

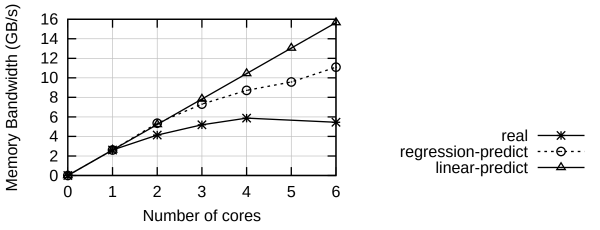


Figure 1. Memory bandwidth usage of *facesim* on an AMD Opteron 6174 with linear prediction model and a regression-based model.

Unfortunately, the memory bandwidth models used in the previous work have low accuracy. They rely on simple mathematical models or regression analysis, and only consider samples of bandwidth usage for prediction. DRAM contention and DRAM concurrency, as well as other important factors (e.g., program memory behaviors) are overlooked. Figure 1 gives the bandwidth usage of a PARSEC benchmark *facesim* running on an AMD 6-core processor [4], as well as the bandwidth predictions of two popular models. One is a linear model that assumes memory bandwidth usages increase linearly with the number of cores [20, 32, 35]. The other is based on multiple logarithmic and linear regressions [19]. As the figure shows, both models have low accuracy. The linear model has an average accuracy of 14% while the regression-based model has an average accuracy of 44% in this example.

The low accuracy in bandwidth prediction can cause the optimization techniques to perform sub-optimally. For example, when managing the core allocation of *facesim* on the AMD processor, these low accuracy models predict the optimal core allocation to be 6 cores, which performs 8% slower than the actual optimal core allocation, 4 cores [20, 35]. More importantly, running with two more cores wastes energy and reduces system utilization.

To the best of our knowledge, no existing model can provide highly accurate bandwidth usage predictions on real machines, because of four major challenges,

- The first challenge is to correctly predict the contention for DRAM resources from co-running threads. The severity of DRAM contention varies with program behavior in a complicated manner. However, because DRAM contention significantly impacts bandwidth usage, it must be correctly predicted.
- The second challenge is to correctly predict the DRAM concurrency. DRAM requests accessing different banks can be served simultaneously and overlap with each other. This overlapping further complicates the prediction of the latency of DRAM requests.
- The third challenge is to consider the large variety of hardware and software factors that affect bandwidth usage besides contention and concurrency. These factors have to be clearly identified and carefully considered.
- The last challenge is to design a model with low overhead. Some uses (e.g., resource contention management) require a low-overhead model that can be applied during execution to handle dynamic workloads.

To address these challenges, this paper presents DraMon, a highly accurate bandwidth model that considers a wide range of factors. We demonstrate that predicting bandwidth usages requires predicting DRAM contention (e.g., row buffer hit ratio) and DRAM concurrency. We also show that contention and concurrency can be predicted with high accuracy and in short amount of time using probability theory. We implemented two versions of DraMon: DraMon-T, a memory-trace based model, and DraMon-R, a run-time model which uses performance monitoring units (PMUs) as inputs.

When evaluated on a real AMD machine with memory-intensive benchmarks, DraMon shows high accuracy. DraMon-T has an average accuracy of 99.17% for DRAM contention prediction, and an average accuracy of 94.70% for bandwidth prediction. DraMon-R has accuracies of 98.55% and 93.37% for DRAM contention and bandwidth predictions, respectively. Moreover, DraMon-R prediction requires 0.03 seconds, which adds only 0.5% overhead to execution time on average.

DraMon predicts the bandwidth usage of a multi-threaded program when it executes using one memory controller (MC). For multiple MCs (non-uniform memory architecture, NUMA) with no inter-MC memory references, these MCs can be modeled independently using DraMon. The case with inter-MC references is left for future work. To avoid introducing other CPU resource contention and increasing synchronization and context switch overheads, here we do not consider simultaneous multi-threading (SMT) and assume only one application thread is executing on one core.

Our contributions include:

- A memory-trace based model, DraMon-T, which predicts bandwidth usage, DRAM contention and concurrency with high accuracy on a real multi-core system.
- A run-time model, DraMon-R, which does not rely on memory traces, but instead uses PMU readings as inputs. DraMon-R has a similar accuracy as DraMon-T with very low overhead.
- An analysis of the factors related to bandwidth usage and of their usages for bandwidth prediction.

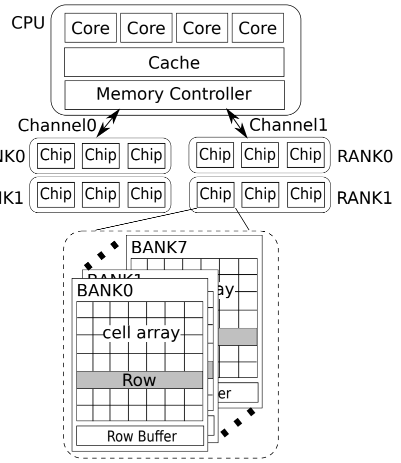


Figure 2. Memory/DRAM systems.

The rest of the paper is organized as follows. Section 2 discusses DRAM systems. Section 3 provides a high level overview of DraMon. Section 4 explains DraMon in detail. Section 5 describes how to obtain input parameters. Section 6 evaluates DraMon on a real machine. Section 7 discusses other related issues. Section 9 discusses related work and Section 10 summarizes the paper.

2. Memory System Background

Before introducing DraMon, this section describes the memory systems on contemporary multi-core platforms.

2.1. DRAM Architecture

Figure 2 depicts a generic memory system and DRAM structure. The on-chip memory controller (MC) is connected to several **channels**. A DRAM request can access one channel at a time, or it can access all channels at once, depending on the configurations of the MC. A channel is composed of several **ranks**. A rank can be roughly viewed as a memory module. Each rank has several memory chips. A memory chip is composed of several **banks**. Each bank is essentially a cell array where a cell is used to store 0 or 1. The banks with the same index of all chips form a conceptual bank of a rank. For example in Figure 2, the $BANK0$s of all chips of $RANK1$ form its conceptual $BANK0$. When $BANK0$ is accessed, the $BANK0$s of every chip on $RANK1$ are activated simultaneously.

2.2. DRAM Request Types

Each bank (both conceptual and physical) has a row buffer. When accessing a piece of data, the row containing this data is read into the row buffer. Then the target data is extracted and sent to MC. Figure 3 shows the operations of a DRAM

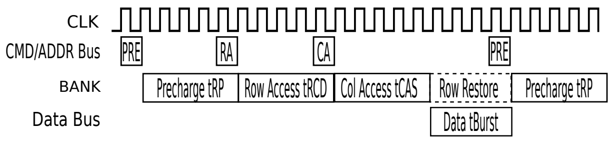


Figure 3. A read cycle.

read [16]. First, the connections between the row buffer and the cell array are precharged (PRE). This precharge is crucial for stable reading, and it requires **tRP** time. Next, the MC issues a row access (RA) command and reads the row into the row buffer in **tRCD** time. After the row is ready, MC sends the column address (CA) and locates the target data in **tCAS** time. Finally, the data is extracted and send to MC using **tBurst** time.

Depending on the status of the target bank, a DRAM request falls into one of three categories [2]:

- **Hit**: the row buffer has the desired row. Only column access and data transportation is required. Therefore, the latency of a hit is $tCAS + tBurst$.
- **Miss**: the bank is precharged, but the row buffer is empty. The desired row has to be read into the row buffer. Therefore, the latency of a miss is $tRCD + tCAS + tBurst$.
- **Conflict**: the bank is not yet precharged for this request. A precharge is required. Therefore, the latency of a conflict is $tRP + tRCD + tCAS + tBurst$.

Note that although a conflict can be viewed as a miss, its service time is very different from a miss. This difference is important for accurately predicting average DRAM request latency and bandwidth usage.

2.3. DRAM Contention

If we compare a row buffer with a cache line, we can easily see similarities. They are both used to temporarily store a copy of data. They are both rewritten to store active data. And they are both shared by and contended for by co-running threads. Consequently, similar to predicting cache contention, which is to predict the hit/miss ratios, predicting DRAM contention is essentially predicting the ratios (percentages) of the three types of DRAM requests: $Ratio_{hit}, Ratio_{miss}$ and $Ratio_{conf}$.

2.4. DRAM Concurrency

The latencies of Section 2.2 are single-request latencies. In practice, multiple DRAM requests can be served simultaneously, thus greatly reducing their average latency. Figure 4 shows four consecutive hits, assuming both $tCAS$ and $tBurst$ are 4 cycles [16]. As the figure shows, the column open operation ($tCAS$) can overlap with data transportation ($tBurst$). Therefore, it only takes 21 cycles to serve all four requests. Thus, the average latency of these hits is only $\frac{21}{4} = 5.25$ cycles, while the full single-request hit latency is $tCAS + tBurst = 8$ cycles.

Similarly, the operations of miss and conflict can also overlap with other DRAM requests. Because this concurrency can

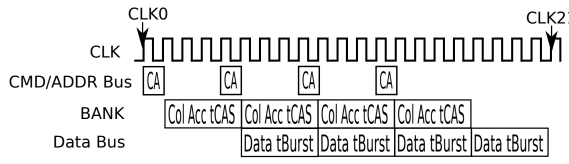


Figure 4. Four concurrent DRAM hits.

significantly reduce average DRAM latency, it must be carefully considered in a bandwidth model.

2.5. Memory Controller Optimizations

Because of the large differences between the latencies of hit, miss and conflict, memory controllers employ several optimization techniques. Accurately predicting memory bandwidth requires considering these optimizations.

The first common optimization is a closed or adaptive page policy, where an opened row buffer is automatically closed and precharged if it is idle for some time [16]. Later, a request to the same bank can then proceed to open a new row without having to wait for precharging. *Four Bank Activation Window (FAW)* also increases the chance of automatic row buffer closing is [16]. Because of power constraints, only four banks can be activated in a rank within a certain time window. This relatively long time window renders opened banks idle and more likely to be automatically closed.

The second common optimization is request reordering [16]. If a request in the MC queue hits an open row, the MC may issue it before the requests that are queued earlier, to enjoy the low latency of DRAM hits.

3. Overview

As stated previously, accurately predicting bandwidth usage requires considering DRAM contention and DRAM concurrency. This section connects DRAM contention and DRAM concurrency to bandwidth usage mathematically, and serves as a road map for the following sections.

3.1. Predicting Bandwidth from DRAM Contention and Concurrency

Memory bandwidth (BW) usage is basically the product of the number of channels *chnl_cnt* available, the memory request rate $Rate_{mem}$ per channel (the number of requests finished per second), and the size of each request $Size_{mem}$:

$$BW = chnl\_cnt \times Rate_{mem} \times Size_{mem}. \quad (1)$$

Memory request rate is determined by two factors: (1) $Rate_{issue}$, the maximum issue rate of DRAM requests limited by program behavior, and (2) $Rate_{dram}$, the DRAM service rate limited by DRAM contention. The actual memory request rate is limited by the smaller of the two:

$$Rate_{mem} = min(Rate_{issue}, Rate_{dram}). \quad (2)$$

Therefore, predicting memory request rate is reduced to the problem of predicting issue rate and DRAM service rate.

Predicting DRAM service rate is equivalent to predicting the reciprocal of the average DRAM request latency

Lat_{dram}.[1] DRAM latency can be further divided into two components, the average read request latency (Lat_r) and the average write request latency (Lat_w). As an optimization, modern MCs delay write requests and group them together for issuing [6, 16]. Because reads and writes are processed separately by MCs, their average latencies can be computed separately. Their weighted average is then the average DRAM latency. Additionally, switching from writes to reads requires stalling the data bus which adds an extra overhead (O_{wtr}). Similarly, when multiple ranks are accessed, rank-to-rank switching also requires data bus stalls and adds overhead (O_{rtr}). Assume the read request ratio is $Ratio_r$ and the write request ratio is $Ratio_w$. Summarizing the above, we have:

$$\begin{aligned} Rate_{dram} &= \frac{1}{Lat_{dram}}, \\ Lat_{dram} &= Ratio_r \times Lat_r + Ratio_w \times Lat_w + O_{wtr} + O_{rtr}. \end{aligned} \tag{3}$$

Similar to predicting the average cache latency, the average read/write latency can be computed using the following equations [13]:

$$\begin{aligned} Lat_r =& Ratio_{hit} \times Lat_{r,hit} + Ratio_{miss} \times Lat_{r,miss} + \\ & Ratio_{conf} \times Lat_{r,conf} \\ Lat_w =& Ratio_{hit} \times Lat_{w,hit} + Ratio_{miss} \times Lat_{w,miss} + \\ & Ratio_{conf} \times Lat_{w,conf}. \end{aligned} \tag{4}$$

Note that the hit/miss/conflict (HMC) ratios here are the average ratios of both reads and writes. The actual ratios of reads and writes are different. However, because their latencies are close (differ by one cycle), using average HMC ratios for both reads and writes is a good approximation.

Combining the above equations gives equation (5) in Figure 5. This equation connects bandwidth usage to DRAM contention (HMC ratios, $Ratio_{ty}$) and DRAM concurrency (HMC latencies, $Lat_{r/w,ty}$). Predicting bandwidth also requires predicting the maximum issue rate $Rate_{issue}$, the write-to-read switching overhead O_{wtr} and the rank-to-rank switching overhead O_{rtr}. Algorithm 1 gives the sections where these components are predicted. Four variables (chn_cnl, $Size_{mem}$, $Ratio_r$, $Ratio_w$) can be acquired from memory traces or PMUs.

3.2. Model Algorithm

Algorithm 1 gives the steps of using DraMon, which requires several parameters as inputs, including parameters that describe a platform's hardware configuration and the parameters that describe a program's memory behavior. With these parameters, DraMon predicts the memory bandwidth for a multi-threaded program running with a certain number of cores/threads using equations (5).

[1]The DRAM latency here is not the single-request latency. It is rather the average latency of multiple overlapped requests, which is the time between they are issued from the MC and the data returned to the MC. It is only used to predict memory bandwidth.

Algorithm 1 Algorithm of DraMon

```
 1: collect hardware related parameters (Sect. 5.2)
 2: for each program $p$ do
 3:     collect software related parameters (Sect. 5.3)
 4:     for each core/thread count $n$ do
 5:         predict maximum issue rate $\mathbf{Rate_{issue}}$ (Sect. 4.1)
 6:         predict HMC ratios $\mathbf{Ratio_{ty}}$ (Sect. 4.2)
 7:         predict HMC latencies $\mathbf{Lat_{r/w,ty}}$ (Sect. 4.3)
 8:         predict write-to-read overhead $\mathbf{O_{wtr}}$ (Sect. 4.4)
 9:         predict rank-to-rank overhead $\mathbf{O_{rtr}}$ (Sect. 4.5)
10:         predict bandwidth usage $\mathbf{BW}$ (Eq. (5))
11:     end for
12: end for
```

4. The Bandwidth Model in Detail

This section discusses the prediction of the unknown components of equation (5) in detail. Note that our prediction equations require several input parameters. Obtaining these parameters is discussed in Section 5.

4.1. Predicting Issue Rate

For a multi-threaded program, if the issue rate of a single core/thread is $Rate_{issue\_single}$ (input parameter obtained in Section 5), then its maximum possible issue rate when running with n cores/threads is

$$Rate_{issue} = Rate_{issue\_single} \times n. \tag{6}$$

4.2. Predicting HMC ratios

Here we describe the prediction of the hit/miss/conflict (HMC) ratios of one thread of a multi-threaded program. The same process can be applied to predict the HMC ratios of others threads. The overall HMC ratios of the program is then the average of all threads' HMC ratios.

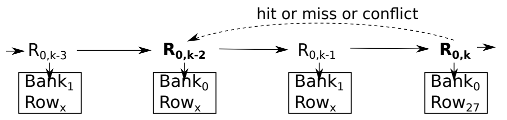
We first use an example with two threads to illustrate the basic idea of our DRAM contention prediction. Then we expand this idea to handle any number of threads and describe the equations for predicting HMC ratios. For simplicity, we focus on predicting hit ratio first.

4.2.1. A Two-thread Example

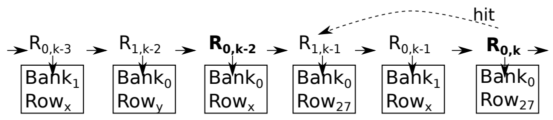
Consider a case where two threads T_0 and T_1 are executing simultaneously. Here we predict the T_0's hit ratio. Naturally, predicting the hit ratio requires identifying the hits in T_0's

$$\begin{aligned} \mathbf{BW} =& chnl\_cnt \times min(\mathbf{Rate_{issue}}, \frac{1}{Ratio_r \times Lat_r + Ratio_w \times Lat_w + \mathbf{O_{wtr}} + \mathbf{O_{rtr}}}) \times Size_{mem} \\ Lat_x =& \sum_{ty} \mathbf{Ratio_{ty}} \times \mathbf{Lat_{x,ty}}, \quad \text{where } x \in \{r, w\}, ty \in \{hit, miss, conf\} \end{aligned} \tag{5}$$

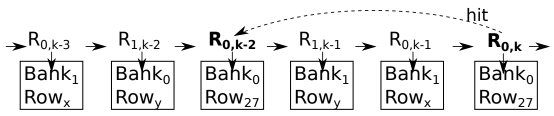
Figure 5. Overall equation for predicting bandwidth usage. Bold faced variables are unknown and require prediction.


hit or miss or conflict
$R_{0,k-3}$ $R_{0,k-2}$ $R_{0,k-1}$ $R_{0,k}$
$Bank_1$ Row_x $Bank_0$ Row_x $Bank_1$ Row_x $Bank_0$ Row_{27}

(a) Requests when there is only T_0 running.


hit
$R_{0,k-3}$ $R_{1,k-2}$ $R_{0,k-2}$ $R_{1,k-1}$ $R_{0,k-1}$ $R_{0,k}$
$Bank_1$ Row_x $Bank_0$ Row_y $Bank_0$ Row_x $Bank_0$ Row_{27} $Bank_1$ Row_x $Bank_0$ Row_{27}

(b) Case 1: Request $R_{0,k}$ hits the row opened by $R_{1,k-1}$.


hit
$R_{0,k-3}$ $R_{1,k-2}$ $R_{0,k-2}$ $R_{1,k-1}$ $R_{0,k-1}$ $R_{0,k}$
$Bank_1$ Row_x $Bank_0$ Row_y $Bank_0$ Row_{27} $Bank_1$ Row_y $Bank_1$ Row_x $Bank_0$ Row_{27}

(c) Case 2: Request $R_{0,k}$ hits the row opened by $R_{0,k-2}$.

Figure 6. Predicting the hit ratio of thread T_0 when it is running with another thread T_1.

requests and computing their percentage. However, this approach requires processing millions of requests which is time consuming. Here we use a key insight we gain into the relation between hit ratio and hit probability.

Insight 1: The hit ratio of T_0 is equivalent to the probability that one of its requests is a hit. Conversely, predicting the hit ratio of T_0 is equivalent to predicting the probability that an arbitrary request of T_0 is a hit.

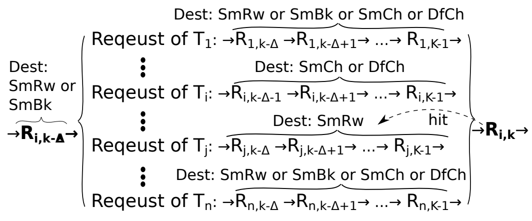
This insight allows us to focus on one single request. It also permits predicting the hit ratio using probability theory which greatly reduces prediction time. Without loss of generality, here we predict the hit probability of the k'th request of T_0, denoted by $R_{0,k}$.

Whether $R_{0,k}$ is a hit, depends on its preceding requests. Figure 6a gives a sequence of requests when T_0 runs alone. The box under each request gives its bank and row addresses. In this figure, $R_{0,k-2}$ is the last request before $R_{0,k}$ that accesses the same bank ($Bank_0$) used by $R_{0,k}$. Depending on $R_{0,k-2}$'s row address, $R_{0,k}$ can be a hit, miss or conflict. If $R_{0,k-2}$ also accesses Row_{27}, $R_{0,k}$ is a hit. If $R_{0,k-2}$ accesses a row other than Row_{27}, $R_{0,k}$ is a conflict. If the row opened by $R_{0,k-2}$ is closed by the MC, then $R_{0,k}$ is a miss.

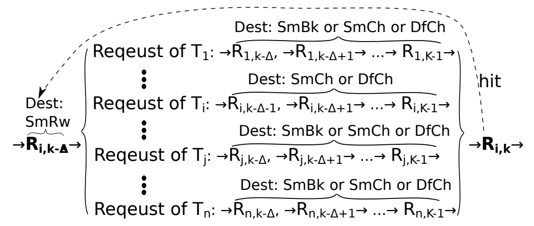
Similarly, when there is co-running thread T_1, the type (hit/miss/conflict) of $R_{0,k}$ depends on its preceding requests from both threads. However, there are billions of requests preceding $R_{0,k}$. It is impossible to consider all of them. Here, we gain our second key insight from Figure 6.

Insight 2: Only requests issued after $R_{0,k-2}$ (including $R_{0,k-2}$) have to be considered when predicting the type of $R_{0,k}$, because any change made to $Bank_0$'s row buffer by requests before $R_{0,k-2}$ is reset by $R_{0,k-2}$.

This insight greatly reduces the number of preceding requests that require consideration. Figure 6b and 6c show the requests from both T_0 and T_1. In these two figures, only one of T_1's requests, $R_{1,k-1}$, is issued between $R_{0,k-2}$ and $R_{0,k}$, and its destination affects the type of $R_{0,k}$.


Dest: SmRw or SmBk or SmCh or DfCh
Reqeust of T_1: $\to R_{1,k-\Delta} \to R_{1,k-\Delta+1} \to \ldots \to R_{1,K-1} \to$
Dest: SmRw or SmBk
Dest: SmCh or DfCh
Reqeust of T_i: $\to R_{i,k-\Delta-1} \to R_{i,k-\Delta+1} \to \ldots \to R_{i,K-1} \to$
$\to R_{i,k-\Delta} \to$
Dest: SmRw
hit
$\to R_{i,k} \to$
Reqeust of T_j: $\to R_{j,k-\Delta} \to R_{j,k-\Delta+1} \to \ldots \to R_{j,K-1} \to$
Dest: SmRw or SmBk or SmCh or DfCh
Reqeust of T_n: $\to R_{n,k-\Delta} \to R_{n,k-\Delta+1} \to \ldots \to R_{n,K-1} \to$

(a) Case 1: Request $R_{i,k}$ is a hit when T_j's requests access the same row used by $R_{i,k}$.


Dest: SmBk or SmCh or DfCh
Reqeust of T_1: $\to R_{1,k-\Delta}, \to R_{1,k-\Delta+1} \to \ldots \to R_{1,K-1} \to$
hit
Dest: SmCh or DfCh
Dest: SmRw
Reqeust of T_i: $\to R_{i,k-\Delta-1}, \to R_{i,k-\Delta+1} \to \ldots \to R_{i,K-1} \to$
$\to R_{i,k-\Delta} \to$
Dest: SmBk or SmCh or DfCh
$\to R_{i,k} \to$
Reqeust of T_j: $\to R_{j,k-\Delta}, \to R_{j,k-\Delta+1} \to \ldots \to R_{j,K-1} \to$
Dest: SmBk or SmCh or DfCh
Reqeust of T_n: $\to R_{n,k-\Delta}, \to R_{n,k-\Delta+1} \to \ldots \to R_{n,K-1} \to$

(b) Case 2: Request $R_{i,k}$ is a hit when $R_{i,k-\Delta}$ accesses the same row used by $R_{i,k}$.

Figure 7. Predicting the hit ratio of thread T_i when it is running with $n-1$ threads.

There are two cases where $R_{0,k}$ is a hit. In case 1 (Figure 6b), $R_{1,k-1}$ accesses the same row (Row_{27}) used by $R_{0,k}$. Therefore, $R_{0,k}$ hits the row opened by $R_{1,k-1}$. In case 2 (Figure 6c), $R_{1,k-1}$ does not access the same row (Row_{27}) used by $R_{0,k}$. However, $R_{0,k-2}$ accesses Row_{27}. Therefore, $R_{0,k}$ hits the row opened by $R_{0,k-2}$. If the probabilities of case 1 and case 2 are P_1 and P_2, then the probability that $R_{0,k}$ is a hit is $P_1 + P_2$, which is also the hit ratio of T_0.

In summary, the two key insights above allow us to focus on one request and a limited number of its preceding requests. By enumerating the destinations of the preceding requests, we list all the cases which can produce a hit. Then we predict the probabilities of these cases, the sum of which is then the hit ratio. Next we generalize this example to any number of threads, and predict the case probabilities.

4.2.2. Generalizing to n threads

Consider the case where n threads are running simultaneously. Here we predict the hit ratio of the i'th thread T_i. According to Insight 1, predicting T_i's hit ratio is equivalent to predicting the probability that its k'th request $R_{i,k}$ is a hit.

Assume the nearest preceding request from T_i that accesses the same bank used by $R_{i,k}$ is $R_{i,k-\triangle}$. Insight 2 can be generalized as: only requests after $R_{i,k-\triangle}$ (including $R_{i,k-\triangle}$) should be considered when determining the type of $R_{i,k}$. Moreover, we define the **Bank Reuse Distance (BRD)** of $R_{i,k}$ as Δ. In Figure 6, $R_{0,k}$'s BRD is 2 because the nearest same-bank accessing request from T_0 is $R_{0,k-2}$.

Figure 7a gives the sequence of requests issued between $R_{i,k-\triangle}$ and $R_{i,k}$ by all n threads. Each row between $R_{i,k-\triangle}$ and $R_{i,k}$ represents the requests issued from one thread. Threads are depicted in independent rows because they execute in parallel. The type (hit/miss/conflict) of $R_{i,k}$ depends on preceding requests' destinations, which fall into four categories:

- $SmRw$, the same row used by $R_{i,k}$.
- $SmBk$, a different row on the bank used by $R_{i,k}$.
- $SmCh$, a different bank of the channel used by $R_{i,k}$.
- $DfCh$, a different channel than $R_{i,k}$'s channel.

To reduce computation time, we assume that all requests from the same thread have the same destination, i.e., the same row. Because of the data locality, this assumption holds for most programs (more than 85% consecutive requests of our benchmarks hit the same row). In Figure 7, the destination of one thread is marked above its requests.

Similar to the example in Figure 6, there are two cases where $R_{i,k}$ is a hit. In case 1 (Figure 7a), at least one thread T_j has a destination of $SmRw$, and $R_{i,k}$ hits the row opened by T_j's requests. In case 2 (Figure 7b) none of the middle threads accessing $SmRw$. However, $R_{i,k-\Delta}$ accesses this row, and $R_{i,k}$ hits the row opened by $R_{i,k-\Delta}$.

The hit probability of $R_{i,k}$, which is also the hit ratio of T_i, is the sum of the probabilities of these two cases. Additionally, the total number of preceding requests have to be determined. Too many preceding requests may cause the MC to close the bank accessed by $R_{i,k}$ (recall the adaptive page policy). Next we discuss predicting these values.

4.2.3. Predicting the Number of Preceding Requests

From BRD's definition, there are Δ requests of T_i that should be considered when predicting the type of $R_{i,k}$.

For a co-running thread T_j, the number of its requests to be considered depends on its issue rate. Assume the single thread issue rate of T_i is $Rate_{issue,i}$, and the single thread issue rate of T_j is $Rate_{issue,j}$. During the time when T_i issued Δ requests, the number of requests issued by T_j is

$$ReqCnt_{\Delta,j} = \Delta \times \frac{Rate_{issue,j}}{Rate_{issue,i}}. \tag{7}$$

4.2.4. Predicting Case Probabilities and the Hit Ratio

Hit ratio prediction requires the following input parameters:

- the hit ratio of T_i when there are no co-running threads, $Ratio_{hit,single}$,
- the probabilities, P_{SmRw}, P_{SmBk}, P_{SmCh} and P_{DfCh}, that a co-running thread access SmRw, SmBk, SmCh and DfCh respectively,
- the number of requests after which an opened row-buffer is automatically closed, D_{ac}, and,
- the total number of threads, n.

Obtaining these parameters is discussed in Section 5. With these inputs, DraMon predicts the probabilities of the two cases in Figure 7 and the hit ratio of T_i.

Case 1: At least one co-running thread has a destination of SmRw (Figure 7a). This case can be further broken down into two sub-cases.

Sub-case A: No co-running thread has a destination of SmBk. The probability of sub-case A is

$$\begin{aligned} P_{caseA} &= P(\exists SmRw \wedge \nexists SmBk) \\ &= P(\nexists SmBk) - P(\nexists SmRW \wedge \nexists SmBk) \\ &= (1 - P_{SmBk})^{n-1} - (P_{SmCh} + P_{DfCh})^{n-1}. \end{aligned} \tag{8}$$

Clearly, sub-case A is a hit:

$$Ratio_{hit,\Delta}(caseA) = P_{caseA}. \tag{9}$$

Sub-case B: At least one co-running thread has a destination of SmBk. The probability of sub-case B is

$$\begin{aligned} P_{caseB} &= P_{case1} - P_{caseA} = P(\exists SmRw) - P(caseA) \\ &= (1 - P(\nexists SmRw)) - P_{caseA} \\ &= (1 - (1 - P_{SmRw})^{n-1}) - P_{caseA}. \end{aligned} \tag{10}$$

Sub-case B can be a hit or a conflict depending on whether the last access before $R_{i,k}$ is a SmRW or SmBk. Because the orders of the requests are random, the type of the last request follows uniform distribution. Therefore, approximately half of the permutations are hits:

$$Ratio_{hit,\Delta}(caseB) = \frac{1}{2} P_{caseB}. \tag{11}$$

Case 2: No co-running thread has a destination of SmRw. However, if $R_{i,k-\Delta}$ is SmRw, $R_{i,k}$ may still be a hit (Figure 7b). $R_{i,k-\Delta}$ is SmRw means $R_{i,k}$ is hit when there are no co-running threads. Therefore, the probability that $R_{i,k-\Delta}$ is SmRw is actually T_i's single thread hit ratio, $Ratio_{hit,single}$.

Case 2 can be broken down into several sub-cases depending on whether a co-running thread accesses the same channel of $R_{i,k}$. We represent each sub-case as a vector. For example, the l'th sub-case is $E_{l,\Delta} = \{e_{l,1}, ..., e_{l,j}, ..., e_{l,n}\}$. An element $e_{l,j}$ represents whether thread T_j accesses $R_{i,k}$'s channel: $e_{l,j} = 1$ means yes, and $e_{l,j} = 0$ means no. Clearly, there are 2^n sub-cases ($1 \leq l \leq 2^n$). In sub-case $E_{l,\Delta}$, the total number of requests from co-running threads that access $R_{i,k}$'s channel is

$$m_{l,\Delta} = \sum\nolimits_j ReqCnt_{\Delta,j} \times e_{l,j}. \tag{12}$$

The probability of sub-case $E_{l,\Delta}$ is then

$$\begin{aligned} P_{l,\Delta} = & Ratio_{hit,single} \times \prod\nolimits_j (e_{l,j} \times (P_{SmBk} + \\ & P_{SmCh}) + (1 - e_{l,j}) \times P_{DfCh}). \end{aligned} \tag{13}$$

If there is no SmBk request, then $E_{l,\Delta}$ can be a hit if the row buffer is not automatically closed. If there are SmBk requests, $E_{l,\Delta}$ can still be a hit if the row buffer is not auto-closed and the MC reorders the requests. In short, $E_{l,\Delta}$ is a hit if the row buffer is not auto-closed:

$$Ratio_{hit,l,\Delta}(case2) = \begin{cases} P_{l,\Delta}, & \text{if } m_{l,\Delta} < D_{ac}, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

The hit ratio is then the sum of all sub-cases:

$$\begin{aligned} & Ratio_{hit,\Delta} = Ratio_{hit,\Delta}(caseA) + \\ & Ratio_{hit,\Delta}(caseB) + \sum\nolimits_l Ratio_{hit,l,\Delta}(case2). \end{aligned} \tag{15}$$

Note that different requests have different BRDs. In other words, for an arbitrary request $R_{i,k}$, it may have different BRDs with different probabilities. Assume the input parameter, the probability of BRD Δ is P_Δ. Then the hit ratio of thread T_i is the sum of the hit ratios of all its BRDs:

$$Ratio_{hit} = \sum_{\Delta} P_\Delta \times Ratio_{hit,\Delta}. \tag{16}$$

4.2.5. Predicting Miss/Conflict Ratios

The miss and conflict ratios can be predicted similarly.

4.3. Predicting Request Latencies

With HMC ratios determined, this section discusses the prediction of HMC latencies. From Figure 4, we gain a third key insight into DRAM concurrency and HMC latencies.

Insight 3: When there is large number of DRAM requests served concurrently, the average latencies of hit/miss/conflict requests are approximately their maximum latencies minus the time that they overlap with other requests' data transfers.

Assume the maximum latencies of hit/miss/conflict are $Max_{ty}, ty \in \{hit, miss, conflict\}$, Insight 3 is essentially

$$Lat_{ty} = Max_{ty} - overlapped\_data\_transfers. \tag{17}$$

The maximum latencies are listed in Section 2.2. Therefore, we only have to determine the $overlapped\_data\_transers$.

4.3.1. Hit Latency (Read)

For a hit, the column access time (tCAS in Section 2.2) can overlap with any request's data transfer. However, its own data transfer ($tBurst$) requires exclusive access to the data bus. Consequently, its average latency is

$$\begin{aligned} Lat_{r,hit} &= Max_{r,hit} - overlapped\_data\_transfers \\ &= (tCAS + tBurst) - tCAS = tBurst. \end{aligned} \tag{18}$$

4.3.2. Miss Latency (Read)

Because a miss opens a new bank, and because of the FAW limit and adaptive page policy, its overlapped data transfer time varies with the type of overlapped requests. That is, we can rewrite equation (17) as

$$Lat_{r,miss} = Max_{r,miss} - \sum_{ty} overlap\_trans\_time_{ty}. \tag{19}$$

Because the overlapped data transfer time is the number of overlapped requests multiplied by the data transportation time ($tBurst$), we can further rewrite the equation to

$$Lat_{r,miss} = Max_{r,miss} - \sum_{ty} overlap\_req_{ty} \times tBurst. \tag{20}$$

Here, the $overlap\_req_{ty}$ represents the number of type ty requests that overlap with one miss request.

Now the problem of determining miss latency is reduced to the problem of determining the number of overlapped requests of each type. First, consider the case of hits overlapping with a miss. Within a sequence of DRAM requests, $Ratio_{hit}$ of them are hits and $Ratio_{miss}$ are misses. Therefore, for one miss, there are at most $\frac{Ratio_{hit}}{Ratio_{miss}}$ hits. Additionally, FAW limits the maximum number of banks ($MaxBk$) that can be simultaneously accessed. Moreover, because concurrent hits are most likely from different threads and do not access the same bank, the total number of concurrent hits is also limited by $MaxBk$. Assume that the input parameter rk_cnt is the number of ranks being accessed. Combining all these arguments, we have

$$\begin{aligned} MaxBk &= rk\_cnt \times 4, \\ overlap\_req_{hit} &= min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{miss}}). \end{aligned} \tag{21}$$

Unfortunately, the number of misses that overlap with another miss cannot be determined using the same approach because there is only one $Ratio_{miss}$. Here, we consider a sequence of n requests from n threads. In this sequence, there are $n \times Ratio_{miss}$ misses. That is, one miss may overlap with $n \times Ratio_{miss} - 1$ misses. Additionally, FAW limit and $MaxBk$ also apply to concurrent miss.

Furthermore, conflicts also require opening new rows, whereas the FAW and adaptive page policy also apply. Therefore, we compute the number of misses and conflicts that overlap with a miss together,

$$\begin{aligned} &overlap\_req_{miss+conf} = min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1). \end{aligned} \tag{22}$$

Combining equations (20) through (22), we have

$$\begin{aligned} &Lat_{r,miss} = (tRCD + tCAS + tBurst) - \\ &(min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{miss}}) + min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1)) \times tBurst. \end{aligned} \tag{23}$$

4.3.3. Conflict Latency (Read)

We predict the average latency of conflicts using the same approach as the miss latency:

$$\begin{aligned} &Lat_{r,conf} = (tRP + tRCD + tCAS + tBurst) - \\ &(min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{conf}}) + min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1)) \times tBurst. \end{aligned} \tag{24}$$

4.3.4. Write latencies

Write HMC latencies can be predicted similarly using the above equations with two changes. First, one extra DRAM cycle besides $tBurst$ is required for data transfer [16]. Second, write recovery time (tWR) should be used as column access time instead of tCAS.

4.4. Write-to-Read Switching Overhead

When switching from write-to-read, the data bus has to be stalled for tWTR time. Assume the ratio of requests that require a write-to-read switch is $Ratio_{wtr}$ (parameter obtained in Section 5). The write-to-read overhead is

$$O_{wtr} = Ratio_{wtr} \times tWTR. \tag{25}$$

4.5. Rank-to-Rank Switching Overhead

When switching between ranks, the data bus has to be stalled for tRTRS time. Assume the ratio of requests that require a rank-to-rank switch is $Ratio_{rtr}$ (parameter obtained in Section 5). The rank-to-rank overhead is

$$O_{rtr} = Ratio_{rtr} \times tRTRs. \tag{26}$$

At this point we have predicted all unknown variables in equation (5), and DraMon is fully presented.

Parameters	Description	Value
$Size_{mem}$	size of each DRAM request	64 Bytes
tRCD	row activation time	13.5 ns
tCAS	column access time	13.5 ns
tRP	precharge time	13.5 ns
tBurst	data transfer time	6 ns
tWR	write recovery time	15 ns
tRTRS	rank switching time	4.5 ns
tWTR	write-to-read switching time	7.5 ns
chnl_cnt	number of channels	2
bk_cnt	number of banks per rank	8
D_{ac}	row buffer auto-close distance	4 requests

Table 1. Hardware Parameters

Parameters	Description
$Rate_{issue,single}$	single thread issue rate
Δ and P_Δ	Bank reuse distances and their probs
$Ratio_{hit,single}$	single thread hit ratio
$Ratio_{miss,single}$	single thread miss ratio
$Ratio_{conf,single}$	single thread conflict ratio
P_{SmRw}	same-row accessing probability
P_{SmBk}	same-bank-diff-row accessing prob.
P_{SmCh}	diff-bank-same-channel accessing prob.
P_{DfCh}	different channel accessing probability
$Ratio_{wr}$	Write request ratio
$Ratio_{wtr}$	write-to-read switching request ratio
$Ratio_{rtr}$	rank switching request ratio
rk_cnt	number of ranks accessed

Table 2. Software Parameters

5. Obtaining Parameters

As discussed in Section 4, DraMon requires input parameters. This section discusses the collection of these parameters.

5.1. Experimental Platform

To demonstrate parameter collection, we use a machine with an AMD Opteron 6174 Processor. This processor has two dies. Each die has six cores which share one 6MB L3 cache and one MC. Each core has 128KB split L1 Cache and 256KB L2 cache. Each MC is connected to two channels of total 12GB memory which is composed of six Samsung M393B5273CH0YH9 DDR3-1333 memory modules. Because this research focuses on predicting the bandwidth of one MC, here we use one-die/six-core of this processor. The machine is running Linux 2.6.32.

5.2. Hardware Parameters

Hardware Parameters can be collected from data sheets and PCI configurations registers [2, 30]. Table 1 gives the description of the hardware parameters, as well as their values of the AMD Opteron 6174 processor.

5.3. Software Parameters

Table 2 gives a list of software parameters. Here we describe two approaches to collect their values. One uses memory-traces. The other one does not require traces. Instead, it uses PMUs, and it can be applied during execution.

5.3.1. Trace-based Approach

We run each program with one thread and generate its memory trace with Pin [22]. This trace contains the virtual addresses of memory requests, which are translated to physical addresses using the page table exported by Linux kernel. Each physical address is later translated to a DRAM address, which includes the channel, rank, bank, row and column addresses [2]. This translated trace is fed into an in-house cache simulator to generate DRAM requests.

Parameters	PMU
$Rate_{issue,single}$	DRAM_ACCESSES_PAGE, HW_CPU_CYCLES
$Ratio_{hit,single}$	DRAM_ACCESSES_PAGE:HIT
$Ratio_{miss,single}$	DRAM_ACCESSES_PAGE:MISS
$Ratio_{conf,single}$	DRAM_ACCESSES_PAGE:CONFLICT
$Ratio_{wr}$	MEM_CONTROLLER_REQ:WRITE_REQ
$Ratio_{wtr}$	MEM_CONTROLLER_TURN:WRITE_TO_READ
$Ratio_{rtr}$	MEM_CONTROLLER_TURN:CHIP_SELECT

Table 3. Collecting program parameters from PMUs

Benchmarks	P_{SmRw}	P_{SmBk}	P_{DfCh}	BRDs
streamcluster.	0.01%	6.26%	49.67%	1(78%), 8(22%)
facesim	0.00%	5.57%	50.20%	1(90%), 2(10%)
canneal	0.01%	6.20%	50.67%	1(93%), 8(7%)
fluidanimate.	0.01%	6.28%	48.93%	1(78%), 4(22%)

Table 4. P_{SmRw}, P_{SmBk}, P_{Dfch} and BRDs of four PARSEC benchmarks.

We analyze the DRAM request trace to collect bank reuse distances and their probabilities (Δ and P_Δ), as well as $Ratio_{wr}$, $Ratio_{wtr}$, and $Ratio_{rtr}$. To acquire single thread HMC ratios, we feed the trace into an in-house DRAM simulator.

To obtain P_{SmRw}, P_{SmBk}, P_{SmCh} and P_{DfCh}, we run the program with two threads and collect their memory traces using the above approach. Then we run both traces with our DRAM simulator to generate these probabilities.

The missing last parameter is $Rate_{issue,single}$, which can be acquired using the PMU. On the AMD processor we use, this PMU counter is DRAM_ACCESSES_PAGE (Table 3).

5.3.2. Run-time Approach

Most software parameters can also be collected from PMUs. Table 3 gives a list of software parameters and their corresponding PMUs on the AMD processor we used.

Unfortunately, there is no PMU that provides values for P_{SmRw}, P_{SmBk}, P_{SmCh}, P_{DfCh} and bank reuse distances. However, from memory traces, we discovered that most programs share common values for these variables. Table 4 gives the values of these parameters of four PARSEC benchmarks (for BRD values, a $x(y)$ represents a BRD of x with probability y). The table shows that all benchmarks have P_{SmBk} of around 6%. The reason for this similarity is the memory interleaving behavior of the MC. Currently, when allocating memory pages, MC distributes the pages evenly among the banks for better performance. For example, when there is only one conceptual rank of memory used, there are 16 banks involved (8 per channel). Therefore, the probability that one bank is accessed by a thread is $\frac{1}{16}$. And the probability that two threads accessing the same bank is $\frac{1}{16} \cdot \frac{1}{16} \cdot 16 = 6.25\%$. Because of this memory interleaving behavior, we use $\frac{1}{bk\_cnt \cdot rk\_cnt \cdot chnl\_cnt}$ as P_{SmBk}, where rk_cnt can be acquired from the OS. Similarly, we use 0% for P_{SmRw} because two threads rarely access the same row, and 50% for P_{DfCh} because there are two channels. P_{SmCh} is $1 - P_{SmRW} - P_{SmBk} - P_{DfCh}$.

For BRDs, most programs have a BRD of one with a high probability. The reason for this similarity is data locality, i.e., consecutive requests are likely to access the same row. Consequently, we assume a sequential access pattern for DraMon-R. On our machine, the channel-interleaving dictates that every eight consecutive same-row requests start accessing a new channel. That is, among the eight requests, the first request has a BRD of eight with probability $\frac{1}{8} = 12.5\%$; the reset seven requests have a BRD of one with probability $\frac{7}{8} = 87.5\%$. That is, we use $1(87.5\%)$ and $8(12.5\%)$ as BRDs for run-time prediction.

6. Experimental Evaluation

Our goal is to evaluate the accuracy and fidelity of DraMon. Here we use the same platform specified in Section 5.1. Our experiments use 10 benchmarks from PARSEC2.1 and all 10 benchmarks from NPB-OMP3.3.1 [4, 17]. Three PARSEC benchmarks *bodytrack*, *dedup* and *x264* are not used because they are I/O bound and have very limited bandwidth requirement.

Two kernel benchmarks are also considered for their high bandwidth requirements and wide uses: *fft*, a Fast Fourier Transform program [37], and *bw_mem* from lmbench3 benchmark suite [23]. For PARSEC and NPB benchmarks, the largest executable input sets, "native" and "D", are used. All benchmarks are compiled using GCC/GFortran 4.4.3. PARSEC and kernel benchmarks are compiled with O3 optimization flag, and NPB benchmarks are compiled with O1 flag. These benchmarks cover a variety of memory access patterns, including read/write requests, single-bank/multi-bank accesses and streaming/random accesses.

For each benchmark, we predict its HMC ratios and bandwidth usages when it runs with two to six cores/threads using one MC. Then we compare the predicted values with the real values obtained from PMUs (Table 2), and report the accuracy of our predictions. Additionally, we compare DraMon to a state-of-the-art, multiple linear and logarithmic regressions based bandwidth model [19].

We define the bandwidth prediction accuracy as

$$Accuracy_{bw} = 100\% - \left| \frac{BW_{real} - BW_{predicted}}{BW_{real}} \right|. \tag{27}$$

For HMC ratios predictions, we leverage the *multinomial likelihood* L from the likelihood theory [31],

$$D_{KL} = \sum_{ty} Ratio_{ty,real} \times \log_2 \left(\frac{Ratio_{ty,real}}{Ratio_{ty,predicted}} \right), \quad L = 2^{-D_{KL}}, \quad ty \in \{hit, miss, conflict\}. \tag{28}$$

Intuitively, L represents the probability that a model is accurate if the model predicts a probability distribution. Here, we simply refer to L as "HMC ratios prediction accuracy."

Benchmarks	Memory behavior	Accuracy L	
		DraMon-T	DraMon-R
streamcluster	read, single-rank, streaming	99.83%	99.80%
facesim	read, single-rank, streaming	99.34%	98.26%
canneal	read/write, single-rank	98.65%	98.51%
lu.D	read/write, multi-rank	98.50%	97.57%
mg.D	read/write, multi-rank	99.64%	98.64%
sp.D	read/write, multi-rank	99.07%	97.90%
fft	read/write, single-rank, streaming	99.83%	99.80%
bw_mem	read, single-rank, streaming	99.83%	99.80%
Average		99.17%	98.55%

Table 5. DRAM contention (HMC ratios) prediction accuracy.

6.1. DRAM Contention (HMC Ratios) Prediction

Table 5 gives the accuracy of DRAM contention prediction for the eight most memory-intensive benchmarks. As the table shows, DraMon is very accurate for memory-intensive programs with a wide range of memory behaviors.

Currently we collect traces with up to 75 million requests.The recording and processing of a trace takes about 30 minutes. It may be possible to use shorter traces for online processing as suggested by previous research [41].

For the rest benchmarks (other than the eight in Table 5), the average accuracy of DraMon-T is 96.99%, and the average accuracy of DraMon-R mode is 96.85%. The highest accuracy of DraMon-T is 99.53% (*fluidanimate*). The highest accuracy of DraMon-R is 99.48% (*raytrace*). The lowest accuracy of both models is 92.46%(*blackscholes*). Because of space limitations, we cannot present each benchmark's result. The overall accuracies of DraMon-T and DraMon-R for all benchmarks are 97.95% and 97.61%, respectively.

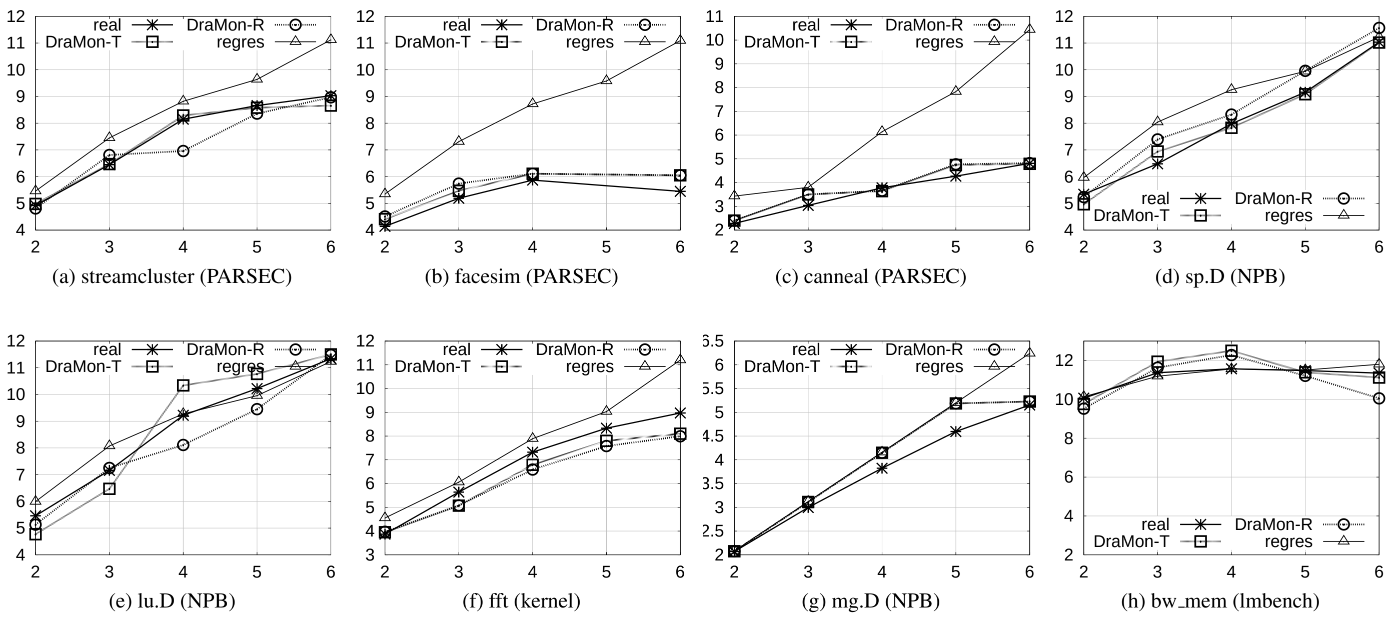
6.2. Bandwidth Usage Prediction

6.2.1. Bandwidth Results

Figure 8 composes the bandwidth prediction results for the eight memory-intensive benchmarks. The result of *facesim* for five cores/threads is missing because it fails to execute with five cores/threads.

DraMon-T has an average accuracy of 94.7%, and DraMon-R has an average accuracy of 93.37%. These results demonstrate that DraMon can accurately predict bandwidth usage for programs with a wide range of bandwidth requirements and memory behaviors. The highest accuracies of DraMon-T and DraMon-R are 98.31% (*streamcluster*) and 95.07% (*mg.D*), respectively. The lowest accuracies of DraMon-T and DraMon-R are 91.49% (*lu.D*) and 90.30% (*fft*), respectively.

The bandwidth usages of the rest benchmarks increase linearly with the number of cores/threads. Both models have a high average accuracy of 97.61%, with the highest accuracy of 99.31% (*blackscholes*) and the lowest accuracy of 90.43% (*bt.D*). The overall accuracies of DraMon-T and DraMon-R for all benchmarks are 96.32% and 95.73%..



(a) streamcluster (PARSEC) (b) facesim (PARSEC) (c) canneal (PARSEC) (d) sp.D (NPB)

(e) lu.D (NPB) (f) fft (kernel) (g) mg.D (NPB) (h) bw_mem (lmbench)

Figure 8. Bandwidth prediction results. X-axis is the number of cores/threads, and the Y-axis is memory bandwidth in GB/s.

6.2.2. Comparing to Regression Model

In Figure 8, results labeled with "regres" are the predictions from a regression-based model [19]. The average accuracy of this model is 77.61%, which is significantly lower than DraMon. Its worst case accuracy is 44.37% (*facesim*), which is also lower than DraMon's lowest accuracy (90.30%). Except for *lu.d* and *bw_mem*, six benchmarks have higher accuracies using DraMon. For *bw_mem*, the regression model has a higher accuracy because it is trained using a micro-benchmark that has a similar behavior as *bw_mem*.

6.3. Execution Time of the Run-time Model

The run-time model requires reading seven PMUs (Table 3). We collect the PMU readings for 0.5 seconds of execution. We implemented DraMon using C. The average time for computing the bandwidth of one core/thread configuration is 0.03 seconds. For each benchmark, five configurations are predicted. Including parameter collection, the total prediction time is 0.65 seconds, which only adds 0.5% to the execution time of our benchmarks in average.

7. Discussion

Prefetcher Impact: In our experiments, the memory prefetcher is enabled. This prefetcher fetches a stream of data from memory if a stride memory access pattern is detected [2]. We observe that this prefetcher has a high prefetching accuracy. It also adaptively decreases the number of prefetching requests in case of heavy DRAM contention [2]. Therefore, the existence of this prefetcher does not affect DraMon's accuracy. However, a less accurate or non-adaptive prefetcher may need to be modeled separately.

DRAM Refresh Impact: DRAM cells need periodical refreshing to retain their data, which can degrade DRAM performance. The DRAM module used in our experiments requires that each bank spend $160ns$ on refreshing every $7.8us$ [30]. Therefore, the DRAM refresh has a theoretical overhead of $\frac{160ns}{7.8us} = 2\%$ [38]. This overhead may be lower than 2% if rank-level parallelism happens [16]. This low overhead does not significantly impact the accuracy of DraMon. Additionally, DRAM refresh overhead can be mitigated for high density DRAM modules [24, 26, 34].

Cache Impact: This research focuses on DRAM, predicting cache performance is beyond its scope. However, because DraMon is evaluated on a real machine, cache does have some impact on our results. Fortunately, memory-intensive benchmarks already have high cache miss rates, and their memory behaviors are not affected by cache contention.[2]

However, four benchmarks, *ferret*, *swaptions*, *freqmine*, and *ep.D* which have very low bandwidth requirements, are affected by cache contention or data sharing. Predicting their bandwidth usages requires a cache model [36, 40]. Because these benchmarks' bandwidth usages depend on cache, their results are not included in the average accuracies in Section 6.

Generalization: Using DraMon on a new platform requires updating its input parameters accordingly. The hardware parameters can be updated based on the new hardware configuration. For DraMon-T, the software parameters can still be obtained from memory traces. For DraMon-R, the corresponding PMUs should be identified on the new platform. Software parameters which cannot be obtained from PMUs

[2]These benchmarks are known as *devils* by previous research [42].

(i.e., SmRw, SmBk, SmCh and DfCh probabilities, and bank reuse distances) are determined by the DRAM configuration (banks/ranks/channels), as well as the channel-interleaving scheme as discussed in Section 5.3.2.

8. Case Study

To demonstrate the performance benefits brought by DraMon, we present a case study where we use both DraMon-R and the regression model to predict the experimentally-optimal core allocation of multi-threaded benchmarks [19]. We use four memory-intensive benchmarks, which are *streamcluster*, *facesim*, *canneal* and *bw_mem*. Clearly, the optimal core allocations of these benchmarks are the core allocations that have the highest bandwidth usages.

Using the AMD Opteron 6174 processor, we first ran each benchmark with one thread to collect input parameters using PMUs. With the PMU readings, we predicted the highest-bandwidth-usage core allocation (i.e., the optimal core allocation) for each benchmark using the two models, and adapted the benchmarks to execute with the two predicted optimal core allocations. Then we compared their performance with the default static core allocation, which uses all available cores. We employed a technique similar to that used by Thread Tailor to support changing thread count during execution [20].

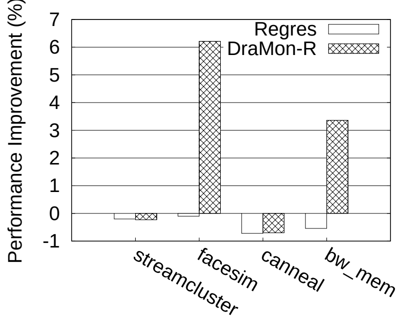


Figure 9. Performance improvement of the optimal core allocations predicted by DraMon-R and the regression model compared to the default static core allocation.

As Figure 8 shows, DraMon-R accurately predicts bandwidth utilization. This accurate prediction yields more accurate core allocation which in turn yields performance improvements. Figure 9 shows that the core allocation based on DraMon-R outperforms the static allocation for *facesim* and *bw_mem*. In contrast, the regression model performs similarly as the static allocation. For *streamcluster* and *canneal*, because their scalability is not bandwidth limited on this particular processor, and because both models have low overhead, all three allocations perform similarly. These results indicate the importance of dynamic core allocation, without which multi-threaded programs may perform sub-optimally and waste energy. These results also show that the benefit of dynamic core allocation can only be achieved with an accurate bandwidth model. Additionally, they show that the low overhead of DraMon adds little performance penalty to dynamic core allocation.

9. Related Work

There are several studies that modeled DRAM analytically. Choi et al., Yuan et al. and Wang modeled the DRAM busy time [7, 38, 44]. These models always require memory traces, and their goal was to provide guidelines for DRAM design rather than predicting bandwidth usage. Kim et al. proposed a model to predict the impact of bank partitioning [18]. Ahn et al. modeled bandwidth usage of programs with regular memory access patterns [1]. These two models do not consider DRAM contention. Additionally, all of these studies were conducted using simulators, while DraMon is evaluated on a real machine.

A Linear bandwidth model was first proposed by Snavely et al. for multi-processors systems [32]. It was later applied to multi-core systems [20, 29, 35]. To improve accuracy, the linear model was also extended with a roof-line, i.e., the maximum peak bandwidth [39]. Kim et al. proposed modeling bandwidth with multiple linear and logarithmic regressions [19]. As demonstrated in this paper, the linear model and regression-based model have low accuracy because several important factors are overlooked.

Cache reuse distance has been used to predict cache miss rate by many studies [5, 9, 40]. These research efforts inspired us to use bank reuse distance.

There also has been work on improving MC/DRAM designs and OS memory allocation algorithms [3, 6, 11, 14, 15, 21, 25, 27, 28, 33]. DraMon can predict the bandwidth usages of these techniques by changing its case selection phase (Section 4.2), or updating its input parameters accordingly.

10. Summary

This paper presents DraMon, a model that predicts the bandwidth usage of multi-threaded programs on real machines with high accuracy. DraMon can be directly employed by previous scalability studies to improve their performance and accuracy. It also can be used to improve DRAM system design and memory allocation algorithms.

We demonstrate that accurately predicting memory bandwidth requires predicting DRAM contention and DRAM concurrency, which both can be predicted with high accuracy and in short computation time using probability theory. We also identify the hardware and software factors that should be considered for bandwidth prediction. These parameters can be collected from memory trace, as well as PMUs for run-time prediction. When evaluated on a real machine, DraMon shows high average accuracies of 98.55% and 93.37% for DRAM contention and bandwidth predictions, with only 0.50% overhead on average.

11. Acknowledgements

This research is supported by National Science Foundation grants CCF-0811689 and CNS-0964627. We appreciate the insightful comments and constructive suggestions from the anonymous reviewers. We would also like to thank Runjie Zhang for his valuable inputs.

References

[1] J. H. Ahn, M. Erez, and W. J. Dally. The Design Space of Data-Parallel Memory Systems. In *Int'l Conf. on Supercomputing*, 2006.

[2] AMD. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, 2013.

[3] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Int'l Symp. on Computer Architecture*, 2012.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. Princeton University, January 2011.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Int'l Symp. on High-Performance Computer Architecture*, 2005.

[6] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads. In *Int'l Symp. on High Performance Computer Architecture*, 2012.

[7] H. Choi, J. Lee, and W. Sung. Memory Access Pattern-Aware DRAM Performance Model for Multi-core Systems. In *Int'l Symp. on Performance Analysis of Systems and Software*, 2011.

[8] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Int'l Symp. on Microarchitecture*, 2010.

[9] C. Ding and Y. Zhong. Predicting Whole-program Locality through Reuse Distance Analysis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2003.

[10] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. The Kremlin Oracle for Sequential Code Parallelization. *IEEE Micro*, 32(4), 2012.

[11] S. Ghose, H. Lee, and J. F. Martínez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *Int'l Symp. on Computer Architecture*, 2013.

[12] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. In *Int'l Symp. on Parallelism in Algorithms and Architectures*, 2010.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[14] E. Herrero, J. Gonzalez, R. Canal, and D. Tullsen. Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments. *IEEE Trans. on Computers*, 62(9), 2013.

[15] M. M. Islam and P. Stenstrom. A Unified Approach to Eliminate Memory Accesses Early. In *Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2011.

[16] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[17] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NASA Ames Research Center, 1999.

[18] D. U. Kim, S. Yoon, and J. W. Lee. An Analytical Model to Predict Performance Impact of DRAM Bank Partitioning. In *Workshop on Memory Systems Performance and Correctness*, 2013.

[19] M. Kim, P. Kumar, H. Kim, and B. Brett. Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model. In *Int'l Symp. on Parallel and Distributed Processing Symposium*. IEEE, 2012.

[20] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *Int'l Symp. on Computer Architecture*, 2010.

[21] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.

[23] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996.

[24] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *Int'l Symp. on Computer Architecture*, 2013.

[25] J. Mukundan and J. F. Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *Int'l Symp. on High Performance Computer Architecture*, 2012.

[26] P. Nair, C.-C. Chou, and M. K. Qureshi. A Case for Refresh Pausing in DRAM Memory Systems. In *Int'l Symp. on High Performance Computer Architecture*, 2013.

[27] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-core, Multi-bank Systems. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* , 2013.

[28] M. K. Qureshi and G. H. Loh. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Int'l Symp. on Microarchitecture*, 2012.

[29] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth wall: Challenges in and Avenues for CMP Scaling. In *Int'l Symp. on Computer Architecture*, 2009.

[30] Samsung. 240pin Registered DIMM based on 2Gb C-die.

[31] J. Shlens. Notes on Kullback-Leibler Divergence and Likelihood Theory. *Systems Neurobiology Laboratory*, 2007.

[32] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *Int'l Conf. on Supercomputing*, 2002.

[33] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *Int'l Symp. on Computer Architecture*, 2010.

[34] J. Stuecheli, D. Kaseridis, H. C. Hunter, and L. K. John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *Int'l Symp. on Microarchitecture*. IEEE, 2010.

[35] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-

driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[36] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[37] R. Thomas and K. Yelick. Efficient FFTs on IRAM. In *Workshop on Media Processors and DSPs*, 1999.

[38] D. T. Wang. *Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm*. University of Maryland, 2005.

[39] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4), Apr. 2009.

[40] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time Modeling of Program Working Set in Shared Cache. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[41] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache Conscious Task Regrouping on Multicore Processors. In *Int'l Symp. on Cluster, Cloud and Grid Computing*, 2012.

[42] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[43] D. Xu, C. Wu, and P.-C. Yew. On Mitigating Memory Bandwidth Contention Through Bandwidth-aware Scheduling. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2010.

[44] G. L. Yuan and T. M. Aamodt. A Hybrid Analytical DRAM Performance Model. In *Workshop on Modeling, Benchmarking and Simulation*, 2009.