A Hybrid Analytical DRAM Performance Model

George L. Yuan Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
{gyuan,aamodt}@ece.ubc.ca

Abstract

As process technology scales, the number of transistors that can fit in a unit area has increased exponentially. Processor throughput, memory storage, and memory throughput have all been increasing at an exponential pace. As such, DRAM has become an ever-tightening bottleneck for applications with irregular memory access patterns. Computer architects in industry sometimes use ad hoc analytical modeling techniques in lieu of cycle-accurate performance simulation to identify critical design points. Moreover, analytical models can provide clear mathematical relationships for how system performance is affected by individual microarchitectural parameters, something that may be difficult to obtain with a detailed performance simulator. Modern DRAM controllers rely on Out-of-Order scheduling policies to increase row access locality and decrease the overheads of timing constraint delays. This paper proposes a hybrid analytical DRAM performance model that uses memory address traces to predict the DRAM efficiency of a DRAM system when using such a memory scheduling policy. To stress our model, we use a massively multithreaded architecture based upon contemporary GPUs to generate our memory address trace. We test our techniques on a set of real CUDA applications and find our hybrid analytical model predicts the DRAM efficiency to within 15.2% absolute error when arithmetically averaged across all applications.

1. INTRODUCTION

As the gap between memory speed and microprocessor speed increases, efficient dynamic random access memory (DRAM) system design is becoming increasingly important as it becomes an increasingly limiting bottleneck. This is especially true in General Purpose Applications for Graphics Process Units (GPGPU) where applications do not necessarily make use of caches (which are both limited in size and capabilities [1] relative to the processing throughput of GPUs).

Modern memory systems can no longer be treated as having a fixed long latency. Figure 1 shows the measured latency for different applications when run on our performance simulator, GPGPU-Sim [3], averaged across all memory requests. As shown, average memory latency can vary greatly across different applications, depending on their memory ac-

[1]As of NVIDIA's CUDA Programming Framework version 2, all caches are read-only [17].

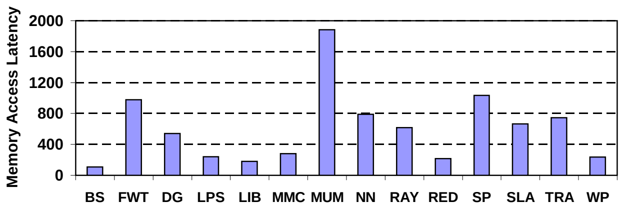


Figure 1: Average memory latency measured in performance simulation

cess behavior. In the massively multithreaded GPU Compute architecture that we simulate, threads are scheduled to Single-Instruction Multiple-Data pipelines in groups of 32 called *warps*, each thread of which is capable of generating a memory request to any location in GPU main memory. (This can be done as long as the limit on the number of in-flight memory requests per GPU shader core has not been reached. In CPUs, this is determined by the number of Miss Status Holding Registers (MSHR) [11].) In this microarchitecture, a warp cannot proceed as long as *any* single thread in it is still blocked by a memory access, making DRAM an even bigger bottleneck. As such, DRAM must be properly modeled to provide useful insight for microprocessor and memory designers alike.

Few analytical models for DRAM have been proposed before [2, 23], with Ahn et al.'s being the most recent [2]. Their first-order analytical DRAM model predicts the achievable bandwidth given a uniform memory access pattern with a constant fixed row access locality. (Row access locality is defined as the number of requests serviced in a DRAM row before the row is closed and a new row is open. More details on modern DRAM systems are given in Section 2.) While this model does not try to handle any non-regular memory access behavior (behavior exhibited in our applications), it does show good accuracy for the microbenchmarks it considers, indicating that analytical modeling of DRAM in more demanding scenarios is promising.

In this paper, we introduce a hybrid analytical DRAM performance model that predicts *DRAM efficiency*, which is closely related to *DRAM utilization*. DRAM utilization is the percentage of time that a DRAM chip is transferring data across its data pins over a given period of time. DRAM efficiency is the percentage of time that a DRAM chip is transferring data across its data pins over a period of time in which the DRAM is not idle. In other words, only time when DRAM has work to do is considered when calculating DRAM efficiency. In essence, both these metrics give a notion of how much useful work is being done by DRAM. Our analytical model is "hybrid" [22] in the sense that it uses equations derived analytically in conjunction with a highly simplified simulation of DRAM request scheduling.

In order to keep up with high memory demands, modern DRAM chips are banked to increase efficiency by allowing requests to a ready bank to be serviced while other banks are unavailable. To use these banks more effectively, modern memory controllers re-order the requests waiting in the memory controller queue [19]. Modeling the effects of the memory controller is thus crucial to accurately predicting DRAM efficiency.

This paper makes the following contributions:

- It presents a novel DRAM hybrid analytical model to model the overlapping of DRAM timing constraint delays of one bank by servicing requests to other banks, given an aggressive First-Ready First-Come First-Serve [19] (FR-FCFS) memory scheduler that re-orders memory requests to increase row access locality.
- It presents a method to use this hybrid analytical model to predict DRAM efficiency over the runtime of an application by profiling a memory request address trace. By doing so, it achieves an arithmetic mean of the absolute error across all benchmarks of 15.2%[2].
- It presents data for a set of applications simulated on a massively multithreaded architecture using GPGPU-Sim [3]. To the best of our knowledge, this is the first work that attempts to use an analytical DRAM model to predict the DRAM efficiency for a set of real applications.

The rest of this paper is organized as follows. Section 2 reviews the mechanics of the GDDR3 memory [21] that we are modeling. Section 3 introduces our first-order hybrid analytical DRAM model, capable of predicting DRAM efficiency for an arbitrary memory access pattern, and describes the two different heuristics of the algorithm we use in our results. Section 4 describes the experimental methodology and Section 5 presents and analyzes our results. Section 6 reviews related work and Section 7 concludes the paper.

2. BACKGROUND

Before explaining the details of our techniques introduced in Section 3, it is necessary to be familiar with the background of modern DRAM and the mechanics of the memory that we are modeling, GDDR3 [21], including the different timing constraints and various other terminology. A summary is provided in Table 2.

GDDR3-SDRAM is a modern graphics-specific DRAM architecture that improves upon older DRAM technology used in graphics processors, like DDR-SDRAM [16], by increasing clock speeds and lowering power requirements. Like DDR-SDRAM, it uses a double data rate (DDR) interface, meaning that it transfers data across its pins on both the positive and negative edge of the bus clock. More specific to the particular GDDR3 technology that we study, a 4n-prefetch architecture is used, meaning that a single read or write access consists of 4 bits transferred across each data pin over 2 cycles, or 4 half-cycles. (This is more commonly known as the burst length, BL.) For a 32-bit wide interface, this translates to a 16 byte transfer per access for a single GDDR3 chip.

[2]In this paper, we use arithmetic mean of the absolute value of error to validate the accuracy of our analytical model, which was argued by Chen et al. [6] to be the correct measure since it always reports the largest error numbers and is thus conservative in not understating said errors.

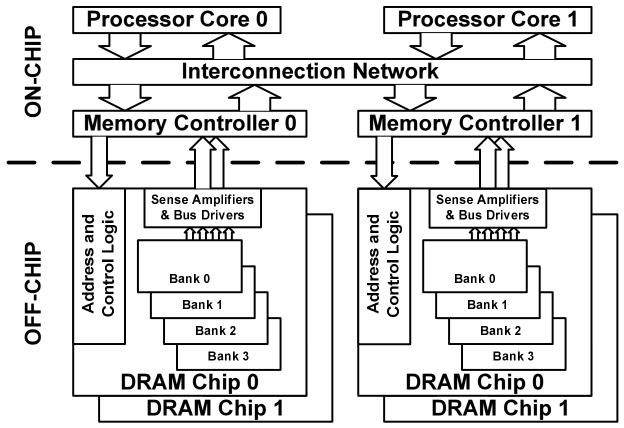


Figure 2: Memory system organization

Similar to other modern DRAM technology, GDDR3-SDRAM is composed of a number of memory banks. Each bank is composed of a 2D array that is addressed with a row address and a column address, both of which share the same address pins to reduce the pin count. In a typical memory access, a row address is first provided to a bank using an activate command that activates all of the memory cells in, or "opens", the row, connecting them using long wires to the sense amplifiers, which are subsequently connected to the data pins. A key property of the sense amplifiers is that, once they detect the values of the memory cells in a row, they hold on to the values so that subsequent accesses to the row do not force the sense amplifiers to re-read the row. This property allows intelligent memory controllers to schedule requests out of order to take advantage of this "row access locality" (as defined in Section 1) to improve efficiency. The organization of such a memory system is shown in Figure 2.

The process of "opening the row," which must be done before a column can be requested, takes a significant amount of time called the row address to column address delay, or t_{RCD} [21]. After this delay, the column address can then be provided to the bank. The process of reading the column address and choosing the data in the specific column to be output across the data pins is not instantaneous, instead taking an amount of time called the column access strobe latency, or CL, from when the column address is provided to when the first bits of data appear on the data pins. However, since all of the memory cells in the opened row are connected to sense amplifiers, multiple column addresses can be provided back-to-back, limited only by the column to column delay, t_{CCD}, and the operations will be pipelined. Once the initial CL cost has been paid, a bank can provide a continuous stream of data as long as all accesses are to the opened row.

If we want to access data in one row and another row is opened in the same bank, we must close the opened row by disconnecting it from the long wires connecting it to the sense amplifiers. Before these long wires, which represent a significant capacitance, can be connected to the new row, they must first be precharged in order for the sense amplifiers to work properly. This process is defined as the row precharge delay, or t_{RP} [7]. To summarize, in order to service a request to a new row when a different row has already been opened, we must first close the row (by issuing a precharge command) and open the new row (by issuing an activate). This process takes an amount of time t_{RP} + t_{RCD}.

Furthermore, in a single bank, there is a minimum time required between issuing an activate command and then issuing a precharge (in other words, between opening and closing the row). This minimum time is defined as the row access strobe latency, t_{RAS}, and is greater than t_{RCD}. This is because t_{RAS} also encapsulates the process of restoring the data from the sense amplifiers to the corresponding row cells, or "refreshing." (Unlike static RAM, which retains the values in its memory cells as long as it is provided power, dynamic RAM must be periodically refreshed since the bit values are stored in capacitors which leak charge over time [7].) Opening a new row too quickly may not allow adequate time to perform the refresh [23]. Accordingly, the minimum time required between issuing two successive activate commands in a single bank is $t_{RAS} + t_{RP}$, which is defined as t_{RC}.

In addition to these timing constraints, there is also a t_{RRD}, the minimum time needed between successive activate commands to different banks, and t_{WTR} and t_{RTW}, the minimum time needed between a write and a read command or a read and a write command, respectively, since the data bus is bi-directional and requires time to switch modes.

In the specific GDDR3 technology that we study, each row must be refreshed every 32ms, taking 8192 DRAM cycles each time. With an 800MHz DRAM clock and 4096 rows per bank, this amounts to each bank having to perform refresh 4.2% of the time on average, assuming DRAM is idle and all rows contain data [21]. As mentioned previously, the process of accessing a row also refreshes the data in the row, meaning only rows of unaccessed data needs to be actively refreshed, possibly reducing the refreshing overhead from 4.2%. Due to this relatively small overhead, we model the effects of refresh on performance in neither our performance simulator nor our analytical model.

3. MODELING GDDR3 MEMORY

In this section, we describe our analytical model of GDDR3 memory. First, we present a more formal definition of DRAM efficiency in Section 3.1. Then, we present our baseline analytical model for predicting DRAM efficiency in Section 3.2. Finally, we show how to implement our model using a sliding window profiling technique on a memory request address trace in Section 3.3.

3.1 DRAM Efficiency

We first define *DRAM utilization* as the percentage of time that a DRAM chip is transferring data across its bus over an arbitrary length of time, T. We also define *DRAM efficiency* as the percentage of time that the DRAM chip is transferring data across its bus over only the period of time when the DRAM is active (T_{active}). We say that a DRAM chip is active when it is either actively transferring data across its bus or when there are memory requests waiting in the memory controller queue for this DRAM and it is waiting to service the requests but cannot due to any of the timing constraints mentioned in Section 2. If the DRAM is always active, the DRAM utilization and DRAM efficiency metric will evaluate to the same value for the same period of time (e.g. T = T_{active}). While 100% DRAM efficiency and utilization is theoretically achievable, less-than-maximum throughput can occur for a variety of different reasons: there may be poor row access locality among the requests waiting in the memory controller queue, resulting in DRAM having to pay the overhead cost of constantly closing and reopening different rows too frequently, and therefore the few requests per open row are not enough to hide the aforementioned overheads; the memory controller may also be causing the DRAM to frequently alternate between servicing reads and writes, thus having to pay the overhead cost of t_{WTR} and t_{RTW} each time; and, specific only to DRAM utilization, the memory controller queue may simply be empty, forcing the DRAM to sit idle.

3.2 Hybrid Analytical Model

As described in the previous section, both our DRAM utilization and DRAM efficiency metrics are essentially fractions. We use these definitions as the basis of our analytical model, replacing the numerators and denominators with an expression of variables that can be derived by analyzing a collected memory request address trace.

We develop our analytical model by first considering requests to a single bank j for a time period T, which starts from when a request to bank j must open a new row and ends when the last request to the new row in bank j present in the memory controller queue at the start of the period has been serviced. We use i to denote the requests to any bank of B banks, not just those to bank j.

$$Eff_j = \frac{MIN\left[MAX(t_{RC}, t_{RP} + t_{RCD} + t_j), \sum_{i=0}^{B-1} t_i\right]}{MAX(t_{RC}, t_{RP} + t_{RCD} + t_j)} \tag{1}$$

where

$$t_i = number\_of\_requests\_to\_bank\_i * T_{srvc\_req} \tag{2}$$

and

$$T_{srvc\_req} = \frac{request\_size}{DRAMs\_per\_MC * busW * data\_rate} \tag{3}$$

To aid our description, we provide the example shown in Figure 3 which will be referred to as we elaborate our model.

The principle underlying Equation 1 is that a bank must first wait at least t_{RC} before switching to a new row. This overhead can be hidden if there are requests to other banks

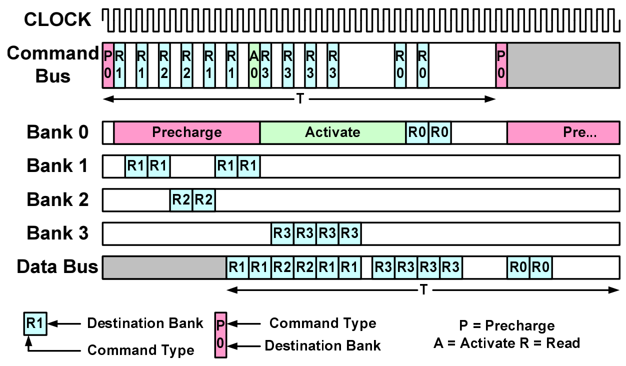


Figure 3: Timing diagram example

that are to opened rows. In Figure 3, this is the case. In this example, j is 0 and the requests to banks 1, 2, and 3 help hide the precharge and activate delays.

When DRAM efficiency is less than 100%, the numerator in Equation 1 is the sum of all t_i, where t_i is defined as the product of two terms: the number of requests that can be serviced by bank i over the set period of time defined in the denominator *multiplied* by the time it takes to service each request (T_{srvc_req}). Note that the sum of all t_i also includes t_j. We assume that all memory requests are for the same amount of data. The formula for calculating how much a single request to bank i contributes to t_i is shown in Equation 3. Here, $request\_size$ is the size of the memory request in bytes, $DRAMs\_per\_MC$ is the number of DRAM chips controlled by each memory controller (e.g., having two DRAM chips connected in parallel doubles the effective bandwidth per memory controller, meaning each single chip only needs to transfer half the data per request), $busW$ is the bus width in bytes, and $data\_rate$ is the number of transfers per cycle, 2 for GDDR3's double data rate interface [21]. Moreover, the minimum granularity of a memory access, $ReqGranularity$ which determines how many read or write commands need to be issued per request, is defined in Equation 4.

$$ReqGranularity = DRAMs\_per\_MC * busW * BL \quad (4)$$

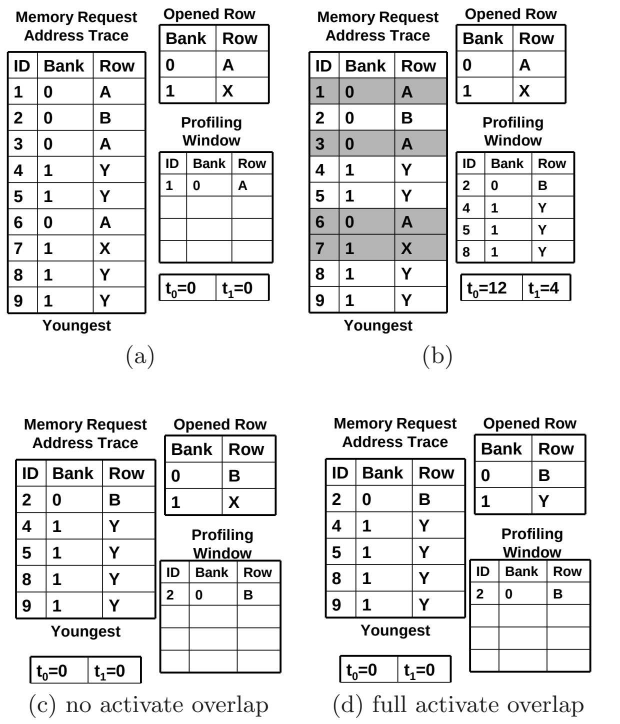
Equation 4 depends on a new parameter unused in the previous equations called the burst length, or BL. Our example corresponds to the baseline configuration described in Section 4, where each request is comprised of 2 read commands to a DRAM chip (2 read commands * 2 DRAM chips per memory controller * 4B bus width * burst length 4 = 64B request size in our baseline configuration).

Basing our model on our defined metrics, we set the denominator of Equation 1 as the period of time starting from when a particular bank of a DRAM chip begins servicing a request to a new row (by first precharging, assuming a different row is opened, and then issuing activate to open the new row) and ending at when the bank issues a precharge to begin servicing a new request to a different row. To simplify our definition of T in this case, our model assumes that a precharge command is not issued until its completion can be immediately followed by an activate to the same bank. Under this assumption, the delay between successive precharge commands is then constrained by the same delay as the delay between successive activate commands, t_{RC}. This is shown as T in Figure 3. The denominator is controlled by two different sets of factors, depending on how many requests need to be serviced in this new row. This is because switching to a new row is primarily constrained by the three timing parameters described in Section 2, t_{RC} (row cycle time), t_{RP} (row precharge delay), and t_{RCD} (row address to column address delay). In the GDDR3 specification that we use, t_{RC} is greater than the sum of t_{RP} and t_{RCD}, so the denominator must always be greater than or equal to t_{RC}. This is expressed in our model using the MAX() function. In our example, since there is only one request (2 read commands) to bank 0, the denominator is dominated by t_{RC} so T equals to t_{RC}.

Given our microarchitectural parameters outlined in Table 1 and Table 2, our read and write requests take four "DRAM clock" cycles to complete, where each request is comprised of two commands. To obtain $number\_of\_requests\_to\_bank\_i$, we count the number of requests waiting in the memory controller queue that are to the row currently open for corresponding bank i. This is equivalent to doing so when the memory controller first chooses the request to the new row of bank j (in other words, right before it needs to precharge and activate). Requests to unopened rows are not counted because they must wait at least t_{RC} before they can begin to be serviced, by which time they can no longer be used to hide the timing constraint overhead for bank j. Since requests to the new row in bank j itself can not be used to hide the latency of precharging and activating the new row, it appears in both the denominator and in the numerator as one of the terms of $\sum_i t_i$ (when $i = j$). In our example, there are 2 read commands to bank 0 and 10 read commands issued to banks 1 through 3 so here t_j=2*2=4 and $\sum_i t_i$ = 2*(2+10) = 24. Therefore, for the time period T, Eff_0 = 24/34 = 70.6%.

As previously described, t_i is used to determine how much of the row switching overhead can be hidden by row-hit requests to other banks. With good DRAM row access locality, there may be more such requests than necessary, causing $\sum_i t_i$ to be greater than the denominator in Equation 1. Using the MIN() function to find the minimum between $\sum_i t_i$ and the expression identical to the denominator is simply done to impose a ceiling on our predicted result at unity, which we argue is accurate since $\sum_i t_i > MAX(t_{RC}, t_{RP}+t_{RCD}+t_j)$ means that there are more than enough requests to completely hide the timing constraint delays imposed by opening the new row in bank j.

As stated before, this equation assumes $T = T_{active}$. In other words, it does not take into account the amount of time when the DRAM is inactive; therefore, it is more suitable for predicting DRAM efficiency rather than DRAM utilization. We expect that provided memory request address traces will not necessarily provide this sort of timing information, in which case we are limited to only being able to predict the DRAM efficiency. In order to find the DRAM efficiency over the entire runtime length of a program, we sum the numerators obtained using Equation 1 of the time periods that make up the runtime length of the program and divide


(a)

Memory Request Address Trace

ID	Bank	Row
1	0	A
2	0	B
3	0	A
4	1	Y
5	1	Y
6	0	A
7	1	X
8	1	Y
9	1	Y

Youngest

Opened Row

Bank	Row
0	A
1	X

Profiling Window

ID	Bank	Row
1	0	A

t_0=0 t_1=0

(b)

Memory Request Address Trace

ID	Bank	Row
1	0	A
2	0	B
3	0	A
4	1	Y
5	1	Y
6	0	A
7	1	X
8	1	Y
9	1	Y

Youngest

Opened Row

Bank	Row
0	A
1	X

Profiling Window

ID	Bank	Row
2	0	B
4	1	Y
5	1	Y
8	1	Y

t_0=12 t_1=4

(c) no activate overlap

Memory Request Address Trace

ID	Bank	Row
2	0	B
4	1	Y
5	1	Y
8	1	Y
9	1	Y

Youngest

Opened Row

Bank	Row
0	B
1	X

Profiling Window

ID	Bank	Row
2	0	B

t_0=0 t_1=0

(d) full activate overlap

Memory Request Address Trace

ID	Bank	Row
2	0	B
4	1	Y
5	1	Y
8	1	Y
9	1	Y

Youngest

Opened Row

Bank	Row
0	B
1	Y

Profiling Window

ID	Bank	Row
2	0	B

t_0=0 t_1=0

Figure 4: Sliding window profiling technique example

the sum by the sum of the corresponding numerators.

$$Eff_j = \frac{\sum_{n=1}^{N} MIN\left[MAX(t_{RC}, t_{RP}+t_{RCD}+t_{j,n}), \sum_{i=0}^{B-1}\right.}{\sum_{n=1}^{N} MAX(t_{RC}, t_{RP}+t_{RCD}+t_{j,n})} \tag{5}$$

Equation 5 shows the DRAM efficiency calculation of a program whose entire runtime length is comprised of N periods, where the DRAM efficiency of each period can be calculated using Equation 1. How we determine which time periods to use is explained next in Section 3.3, where we detail how we use Equation 1 with a memory address trace to determine the DRAM efficiency.

3.3 Processing Address Traces

We assume that the memory request address trace is captured at the point where the interconnection network from the processor feeds the requests into the memory controller queue, allowing the order in which the requests are inserted into the memory controller to be preserved. We assume that the trace is otherwise devoid of any sort of timing information, such as the time when requests enter the queue.

Consider a simple case of a memory controller with a queue size of 4 and a DRAM with only two banks, *Bank 0* and *Bank 1*, where each bank has only two rows (*Row A* and *Row B* for *Bank 0* and *Row X* and *Row Y* for *Bank 1*). Now assume the memory address trace shown in Figure 4(a), where the top request (*Bank 0 Row A*) is the oldest.

To process the trace, we use the algorithm in Figure 5 to determine the t_i values needed for Equation 1. Figure 5 shows two different heuristics which we quantify the accuracy for in Section 5. The first heuristic, which we call *profiling + no activate overlap*, is guarded by the "if(mode == no_activate_overlap)" statement and the second heuristic, which we call *profiling + full activate overlap*, is guarded by the "if(mode == full_activate_overlap)" statement. While this algorithm applies only to First-Ready First-Come First-Serve (FR-FCFS), it can be modified to handle other memory scheduling algorithms. In Section 5.3, we modify our algorithm to predict the DRAM efficiency of another out-of-order scheduling algorithm, "Most-Pending" [19]. Our approach is based on a sliding window profiling implementation where the window is the size of the memory controller queue.

```
Start:  Reset window_size to 0
        while (window_size < memory controller queue size) {
        //while memory controller queue is not full
                Read in newest request, req, to sliding window
                if (req.row == opened_row[req.bank]) {
                //check if request is to opened row, if so, service it
                //(first-ready first-come first-serve policy)
                        remove req from window and from trace
                        $t_{req.bank}$ += $T_{srvc\_req}$//update t
                } else {
                //request is to different row, so store in queue and
                //check if later requests can be serviced first
                        window_size++ //add this req into window
                }
        }
        Calculate_efficiency($t_i$) //uses $t_i$ to calculate efficiency
        Reset($t_i$)
        if (mode == no_activate_overlap){
                //find oldest request, oldest_req
                opened_row[oldest_req.bank] = oldest_req.row
        }
        if (mode == full_activate_overlap){
                //find oldest request of all banks, oldest_req[]
                for (all i in banks) {
                        opened_row[oldest_req[i].bank] =
                                                        oldest_req[i].row;
                }
        }
        Go back to Start
```

Figure 5: Sliding window profiling algorithm for processing memory request address traces

We start with the assumption that initially *Row A* of *Bank 0* and *Row X* of *Bank 1* are open (Figure 4(a)). Following our algorithm, the first request that we encounter is to *Bank 0 Row A*, which is an opened row, so we can remove that request from the sliding window and increment t_0 by T_{srvc_req} (in Section 3.2). The next request that we read in is to the same bank but this time to *Row B*. We leave this request in the sliding window, e.g. our memory controller queue, in the hopes that, by looking ahead to newer requests, we can find ones that are to the opened row. Moving on, it can be seen that the next request is again to *Bank 0 Row A*, allowing us to service it right away.

In this manner, we process all the requests up to Request 8, removing (servicing) requests that are to the opened rows (shown as shaded entries in Figure 4(b)), at which point our window size is 4, meaning the memory controller queue is

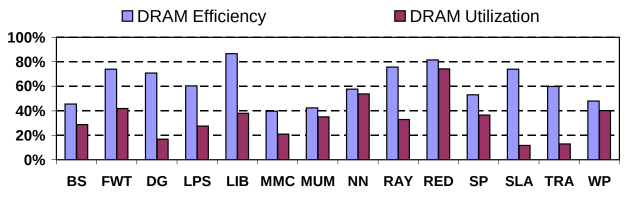


Figure 6: Measured DRAM efficiency and measured DRAM utilization averaged across all DRAM chips for each benchmark

now full. The t_i values can now be used to calculate the efficiency for this period using Equation 1. The t_i values are then reset to signify the start of a new prediction period and repeat our algorithm from the start. For *profiling + no activate overlap*, we then open the row of only the next oldest request, which is to *Bank 0 Row B* (Figure 4(c)). The DRAM efficiency over the entire length of the trace can also be computed using the method described at the end of Section 3.2.

While *profiling + no activate overlap* focuses on the prediction of efficiency by accounting for the requests in the memory controller queue that can help reduce the overhead of timing constraint delays, it does not account for the timing constraints of separate banks that can also be overlapped. More specifically, the process of precharging and activating rows in different banks of the same chip is only constrained by t_{RRD}. With a t_{RC} value of 34 and a t_{RRD} of 8 as per our DRAM parameters 2, we can completely overlap the activation of rows in all 4 banks (34/8 = 4.25 activate commands issuable per row cycle). Performance-wise, this means that even for a memory access pattern with minimal row access locality, the access pattern that finely interleaves requests to all 4 banks can achieve 4 times the bandwidth of an access pattern that has requests to only a single bank. As such, it is crucial to also take into account the overlapping of precharge and activates for applications with poor row access locality. Otherwise, the analytical model will significantly underestimate the DRAM efficiency. We model this using our second heuristic, *profiling + full activate overlap*, which instead opens the rows of the oldest requests to all banks, assuming that the delays associated with switching rows can be overlapped in all banks. This is shown in Figure 4(d) as Bank 1 also switching from Row X to Row Y in accordance with the code guarded by "mode = full_activate_overlap" in Figure 5. Leaving the opened row for Bank 1 as Row X essentially forces the switching overhead of this bank to be paid separately from Bank 0's switching overhead from Row A to Row B (profiling + no activate overlap), potentially causing underestimation of DRAM efficiency. Conversely, switching the rows of all banks at the same time completely overlaps the switching overheads of all banks (profiling + full activate overlap), potentially causing overestimation of DRAM efficiency.

In the next section, we will describe our experimental methodology. In Section 5, we will show the accuracy of the two heuristics described in this section in comparison to the DRAM efficiency measured in performance simulation.

Table 1: Microarchitectural parameters (Bolded Values Show Baseline Configuration)

Shader Cores	**32**
Threads per Shader Core	**1024**
Interconnection Network	**Full Crossbar**
Maximum Supported In-flight Requests per Shader Core	**64**
Memory Request Size (Bytes)	**64**
DRAM Chips	8,**16**,32
DRAM Controllers	**8**
DRAM Chips per Controller	1,**2**,4
DRAM Controller Queue Size	8,16,**32**,64
DRAM Controller Scheduling Policy	**First-Ready First-Come First-Serve (FR-FCFS)**, Most Pending [19]

Table 2: DRAM parameters

Name	Description	Value
	Number of Banks	4
	Bus Width (in Bytes)	4
BL	Burst Length (in Bytes)	16
t_{CCD}	Delay between successive reads or successive writes	2
t_{RRD}	Minimum time between successive ACT commands to different banks	8
t_{RAS}	Minimum time between opening (ACT) and closing (PRE) a row	21
t_{RCD}	Minimum time between issuing ACT and issuing a Read or Write	12
t_{RC}	Minimum time between successive ACTs to different rows in same bank	34
t_{WTR}	Write to Read command delay	5
t_{RP}	Minimum time between closing a row and opening a new row	13
CL	Column Address Strobe Latency	9

4. METHODOLOGY

To evaluate our hybrid analytical model, we modified GPGPU-Sim [3], a massively multi-threaded cycle-accurate performance simulator, to collect the memory request address streams that are fed to the memory controllers for use in our analytical model. Table 1 shows the microarchitectural parameters used in our study. The advantage of studying a massively multi-threaded architecture that GPGPU-Sim is capable of simulating is that it allows us to stress the DRAM memory system due to the sheer number of memory requests to DRAM that can be in-flight simultaneously at any given time. In our configuration, we allow for 64 in-flight requests per shader core. This amounts to a maximum of 2048 simultaneous in-flight memory requests to DRAM. In comparison, Prescott has only eight MSHRs [4] and Williamette has only

Table 3: Benchmarks

Benchmark	Label	Suite
Black-Scholes option pricing	BS	CUDA SDK
Fast Walsh Transform	FWT	CUDA SDK
gpuDG	DG	3rd Party
3D Laplace Solver	LPS	3rd Party
LIBOR Monte Carlo	LIB	3rd Party
Matrix Multiply	MMC	3rd Party
MUMmerGPU	MUM	3rd Party
Neural Network Digit Recognition	NN	3rd Party
Ray Tracing	RAY	3rd Party
Reduction	RED	CUDA SDK
Scalar Product	SP	CUDA SDK
Scan Large Array	SLA	CUDA SDK
Matrix Transpose	TRA	CUDA SDK
Weather Prediction	WP	3rd Party

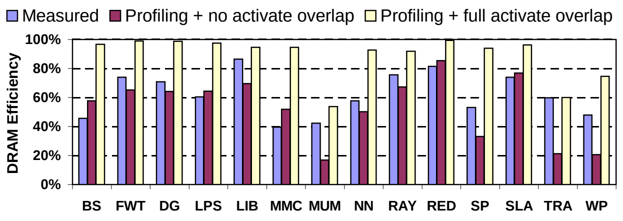


Figure 7: Comparison of measured DRAM efficiency to predicted DRAM efficiency averaged across all DRAM chips for each benchmark

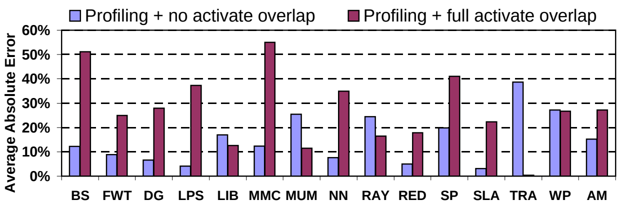


Figure 8: Arithmetic mean of absolute error of predicted DRAM efficiency averaged across all DRAM chips for each benchmark (AM = arithmetic mean across all benchmarks)

four MSHRs [8]. With an eight memory controller configuration, we use our model on 8 separate memory request address streams per application. We test our model on the 14 different applications shown in Table 3, all of which have various levels of measured DRAM efficiency and utilization. We use six applications from NVIDIA's CUDA software development kit (SDK) [15] and seven applications from the set used by Bakhoda et. al in [3]. The last application, Matrix Multiply, is from Ryoo et al [20]. Our only criterium for selecting applications from the two suites is that they must show greater than 10% DRAM utilization averaged across all DRAM chips for our given baseline microarchitecture configuration.

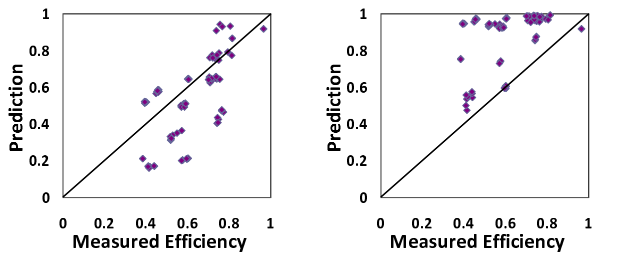
To illustrate the diversity of our application set, Figure 6 shows the DRAM efficiency and utilization measured by GPGPU-Sim in performance simulation averaged across all DRAM chips for each application. We simulate each application to completion in order to collect the full memory request address trace.

5. RESULTS

In Section 5.1, we will first show our results for the two heuristics that we described in Section 3.3. We show that our model obtain a correlation of 68.8% with an arithmetic mean absolute error of 15.2%. In Section 5.2, we provide an in-depth analysis on the cause of our errors. In Section 5.3, we perform a sensitivity analysis of our analytical DRAM model across different microarchitecture configurations.

5.1 DRAM Efficiency Prediction Accuracy

Figure 7 compares the actual measured DRAM efficiency to the modeled DRAM efficiency when using the profiling technique described in Section 3.3. For clarity, we show only the arithmetic average of the efficiency across all DRAM controllers of each application. The arithmetic average of these



(a) Profiling + no activate overlap (b) Profiling + full activate overlap

Figure 9: Comparison of actual DRAM efficiency to modeled DRAM efficiency (1 data point per memory controller per benchmark)

absolute errors is 15.2% for *profiling + no activate overlap* and 27.2% for *profiling + full activate overlap*. The corresponding correlation is 68.8% and 41.6% respectively. We described in Section 3.3 that, without accounting for overlap of row activates, we expect our predictions to underestimate the measured efficiency (*profiling + no activate overlap*). (We explain why the results of some applications do not match these expectations in Section 5.2.) In order to help us determine this, we propose a new metric, *polarity*, which is defined as the arithmetic average of the *non-absolute* errors divided by the arithmetic average of the *absolute* errors. The value of polarity can range between -1 and 1, where -1 means that the test value (predicted efficiency) is always less than the reference value (measured efficiency), 1 means that the test value is always greater than the reference value, and 0 means that on average, the error is not skewed positively or negatively in anyway. We calculated the arithmetic mean of the non-absolute errors for *profiling + no activate overlap* to be -8.8%, meaning a polarity of -0.579. This implies a moderate tendency to underestimate the DRAM efficiency, confirming our expectations. On the other hand, the polarity of average arithmetic error of *profiling + full activate overlap* is 0.985, meaning it almost always overestimates the DRAM efficiency. We expect this to occur since the frequency of issuing activates is constrained by t_{RRD}. Moreover, a row activate to a bank cannot be issued as long as there are still requests to the current row of that bank, further constraining the frequency of issuing activates. Our *profiling + full activate overlap* heuristic captures neither of these, essentially meaning that assuming full overlap of the row activates will always be optimistic in predicting the DRAM efficiency. Figure 9(a) shows the scatter plot of predicted DRAM efficiency using our best profiling method, *Profiling + no activate overlap*, to the measured efficiency and Figure 9(b) shows the scatter plot of our other heuristic, *Profiling + full activate overlap*. Each dot represents the measured DRAM efficiency versus predicted for a single DRAM controller. Several facts become immediately apparent. First of all, the data points in Figure 9(a) are closer to the X=Y line, further illustrating *Profiling + no activate overlap*'s improved accuracy over *Profiling + full activate overlap*. Most of the data points for *Profiling + no activate overlap* tend to be under the X=Y line, meaning that this modeling heuristic tends to underestimate the DRAM efficiency, while *Profiling + full activate overlap* behaves the opposite.

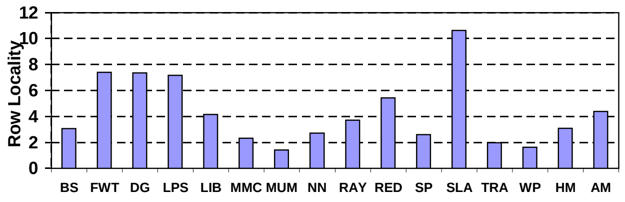


Figure 10: Average row access locality averaged across all DRAM chips for each benchmark (AM = arithmetic mean across all benchmarks, HM = harmonic mean across all benchmarks)

5.2 Modeling Error Analysis

Applications where one technique generates significantly poor predictions tend to be predicted much more accurately by the other technique (BS, FWT, DG, LPS, MMC, MUM, NN, RED, SLA, TRA). It is encouraging to see this because it implies that a heuristic that predicts which to use may reduce the mean absolute error. We postulate that for applications with poor row access locality, the overlapping of row activates has more significant impact. Consequently *profiling + full activate overlap* should perform much better than *profiling + no activate overlap*, which serializes the precharge and activate delays and potentially underestimates the DRAM efficiency. Conversely, we postulate that applications with good row access locality will not benefit as much from row activate overlapping, which will make *profiling + full activate overlap* overestimate the efficiency.

To verify this, we obtain the *average row access locality* by dividing the total number of read and write requests by the number of activates per memory controller and arithmetically averaging these values across all memory controllers for each application. Figure 10 shows this average row access locality. MUM, TRA, and WP exhibit the lowest row access locality and, corresponding with our hypothesis, they are all predicted more accurately by *profiling + full activate overlap* than *profiling + no activate overlap*. A quick calculation showed that the error of a “smart” heuristic that chooses *profiling + full activate overlap* when the average row access locality is less than 2.0 and *profiling + no activate overlap* otherwise would result in an error reduction of 24.8% compared to *profiling + no activate overlap*, from 15.2% to 11.4%. (The value 2.0 was chosen to minimize the error and small variations can increase this error.)

Of the 5 applications where *profiling + full activate overlap* is more accurate (LIB, MUM, RAY, TRA, WP), 3 of them rank in the bottom 3 in terms of row access locality (MUM, TRA, WP). The other 2, LIB and RAY, both have average row access locality, less than the arithmetic mean but more than the harmonic mean. While BS, MMC, NN and SP have row access locality approximately equal to LIB and RAY, *profiling + full activate overlap* predicts less accurately than *profiling + no activate overlap* for these applications by severely overestimating the DRAM efficiency. A closer look at SP shows that it is dominated by reads (99.8% reads and 0.2% writes) and the DRAM is actually stalled during 48% of its total runtime on average due to congestion in the interconnect that sends data from DRAM to the processors. This congestion backs up to the memory controller itself, which cannot issue requests since there is no output buffer space. Therefore, while the DRAM is actually capable of utilizing the bus better and being more efficient, it is unable to due to this interconnect congestion.

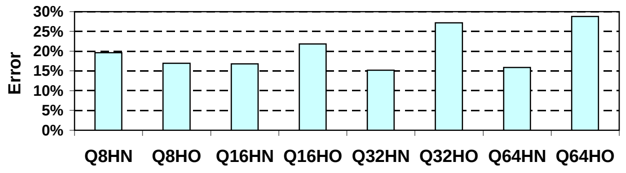


Figure 11: Arithmetic mean absolute error versus DRAM controller queue size averaged across all DRAM controllers and across all benchmarks

For BS and MMC, we noticed that the utilization is quite low, at only 29% and 21% respectively. We measured the average occupancy of the memory controller queue for these applications, considering only the time when there is at least one request waiting in the memory controller queue. Over the course of their entire runtimes, BS and MMC show an average occupancy of only 10% and 19% respectively. (Furthermore, MMC also shows congestion at the interconnect from DRAM to the processors 28% of the time.) This essentially means that our profiling technique is too aggressive in that the sliding window (whose length is equal to the modeled memory controller queue size) that it uses is too large relative to the average memory controller queue occupancy in simulation. As such, the profiling technique is finding row access locality that does not exist in simulations since the majority of the requests have not arrived to the memory controller yet. (Note that the profiling technique used does not take timing information into account so it assumes that the memory controller (sliding window) can always look ahead in the memory request address trace by an amount equal to the memory controller queue size.)

NN is a unique case in that it has four compute kernels that each exhibit different memory access patterns (A compute kernel is defined as a portion of the application capable of having a unique number and organization of parallel threads [17].) The first two compute kernels exhibit high row access locality but, like SP, they suffer from congestion at the interconnect from DRAM to the processor cores, stalling 77% of the time at the interconnect. The row access locality for the third kernel is much lower, as is the occupancy at 37% (compared to 72% for the first two kernels), so it exhibits the same problem as BS and MMC. Both problems cause *profiling + full activate overlap* to overestimate the DRAM efficiency, explaining the large error for this application. (The last compute kernel is significantly shorter than the first three so it contributes little to the average measured and predicted efficiency.)

5.3 Sensitivity Analysis

In this section, we present the accuracy of our results while sweeping across different key parameters: the DRAM controller queue size, the number of parallel DRAM chips per DRAM controller, and the DRAM controller scheduling policy.

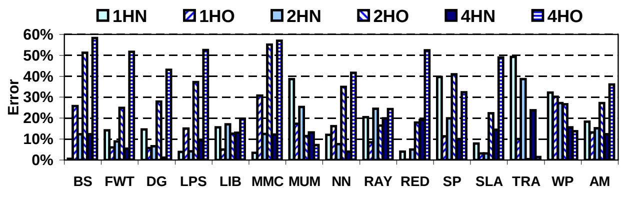


Figure 12: Arithmetic mean absolute error versus number of DRAM chips per DRAM controller (AM = arithmetic mean across all benchmarks)

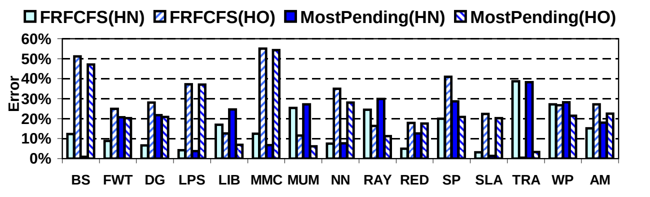


Figure 13: Arithmetic mean absolute error of "Most Pending" memory scheduling algorithm versus FR-FCFS (AM = arithmetic mean across all benchmarks)

We first present Figure 11, which shows the arithmetic mean absolute error of our two profiling heuristics, *profiling + full activate overlap* and *profiling + no activate overlap*, across four different DRAM controller queue sizes of 8, 16, 32 and 64. For clarity, we only present the values averaged across all DRAM controllers and all benchmarks. The naming convention of the different bars is "Q#H*", where Q# is the DRAM controller queue size and H* is the heuristic. HN is *profiling + no activate overlap* and HO is *profiling + full activate overlap*. In general, the error tends to increase as the queue size increases. We attribute this to the fact that, since the queue occupancy is not always 100% for some benchmarks for our baseline DRAM controller queue size of 32 anyways (as explained in Section 5.2), the occupancy will be even lower when the queue size is increased to 64. This means that the size of the profiling window that we use is too large compared to what is occurring in simulation. Moreover, the efficiency of all DRAM controllers decreases as the queue size is reduced, meaning the absolute value of the difference will decrease as well.

Figure 12 shows the modeling error as the number of DRAM chips connected in parallel (to increase bandwidth) to the DRAM controller is varied. The naming convention of the labels for this barchart is "#H*", where the leading # specifies the number of DRAM chips per controller and H* again specifies the two profiling heuristics. Increasing the number of DRAM chips per controller is an effective way of increasing the total off-chip bandwidth to the chip without incurring much additional circuit logic on the chip (although it will increase the chip package size due to an increased number of pins needed for data transfer). Since our memory request data sizes are fixed at 64B and each of our DRAM chips is capable of transferring 16 bytes of data per command, the limit in this amount of parallelism is no more than 4 DRAM chips per controller. Furthermore, as this parallelism is increased, the memory access pattern becomes much more important in determining the DRAM efficiency. This is because more DRAM chips per controller means less read and write commands per request, thus reducing T_{srvc_req}. The number of requests per row (e.g., the row access locality) thus needs to be higher to maintain the DRAM efficiency as shown in Equation 1. Moreover, reducing T_{srvc_req} can reduce the value of the denominator of Equation 1, $MAX(t_{RC}, t_{RP} + t_{RCD} + t_j)$. This means that there is less time to overlap the row activate commands of different banks, implying that *profiling + no activate overlap* should be more accurate than *profiling + full activate overlap*. As we see in Figure 12, this is indeed the case.

Finally, we also try implementing our hybrid analytical model for another memory scheduling policy, "Most Pending" [19]. After all requests to a row have been serviced, our default memory scheduling policy, First-Ready First-Come First-Serve (FR-FCFS) [19], will then open the row of the oldest request. "Most Pending," conversely, will open the row that has the most number of requests pending to it in the queue (e.g., essentially a greedy bandwidth maximization scheme). The algorithm to implement this in our analytical model is virtually the same as the one shown in Figure 5 except, instead of finding the oldest request whenever we switch rows, we now find the oldest request corresponding to the row that has the most number of requests. Figure 13 shows the arithmetic mean absolute error for our baseline configuration. We see that for this scheduling policy, the error is comparable to that of FR-FCFS.

In conclusion, we see that our model is quite robust and the error stays comparable when varying across key microarchitectural parameters and even for a different memory scheduling policy. With smarter heuristics, such as the simple one described in Section 5.2, this error can be reduced even further.

6. RELATED WORK

Ahn et al. [2] explore the design space of DRAM systems. They study the effects on throughput of various memory controller policies, memory controller queue sizes, and DRAM timing parameters such as the Write to Read delay and the Burst Length. More relevant to our work, they also present an analytical model for expected throughput assuming a "random indexed" access pattern and a fixed "record length". This essentially means that they assume a constant fixed number of requests accessed per row and a continuous stream of requests whereas our model handles any memory access pattern. Furthermore, they only show the results of their analytical model for two micro-benchmarks while we test a suite of applications with various memory access patterns. We leave a quantitative comparison between their model and ours to future work.

In his PhD thesis, Wang [23] presents an equation for calculating DRAM efficiency by accounting for the idle cycles in which the bus is not used. His model assumes that the request stream from the memory controller to DRAM is known while we also account for optimizations to the request stream made by the memory controller to help hide timing constraint delays. He also considers only single-threaded workloads, for which he observes high degrees of access locality. Our massively multi-threaded workloads have a wide range of access localities.

There exist many analytical models proposed for a variety of microprocessors [14, 18, 12, 13, 10, 6, 5, 9], from out-of-order execution superscalar to in-order execution fine-grained multithreaded processors to even GPU architectures. Agarwal et al. [1] also present an analytical cache model estimating cache miss rate when given cache parameters such as cache size, block size, associativity, etc., and their model also requires an address trace of the program being analyzed. Of these microprocessor analytical models, only Karkhanis and Smith [10], Chen and Aamodt [6, 5], and Hong and Kim [9] model long, albeit fixed, memory latency.

7. CONCLUSIONS

In this paper we have proposed a novel hybrid analytical DRAM model which takes into account the effects of Out-of-Order memory scheduling and hiding of DRAM timing delays. We showed that this model can be used with a sliding window profiling technique to predict the DRAM efficiency over the entire runtime of an application, given we have its full memory request address trace. We chose a massively multi-threaded architecture connected to a set of high-bandwidth GDDR3-SDRAM graphics memory as our simulation framework and evaluated the accuracy of our model on a set of real CUDA applications with diverse dynamic memory access patterns. Using our sliding window profiling approach with two different heuristics, we were able to predict the DRAM efficiency of a set of real applications to within an arithmetic mean of 15.2% absolute error using our best performing heuristic. We also showed that applications that are predicted poorly by one heuristic tend to be predicted much better with the other, implying the possibility for improvement by combining the two heuristics or develop more intelligent ones, which we leave to future work.

8. ACKNOWLEDGMENTS

We would like to thank Xi Chen, Ali Bakhoda, Johnny Kuan, Andrew Turner, Aaron Ariel, Hyunjung Lee, and the anonymous reviewers for their valuable comments on this work. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[2] J. H. Ahn, M. Erez, and W. J. Dally. The Design Space of Data-Parallel Memory Systems. In *SC 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 80.

[3] A. Bakhoda, G. L. Yuan, W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS 2009: Proceedings of the 2009 Annual IEEE International Symposium on Performance Analysis of Systems and Software.*

[4] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology. *Intel® Technology Journal*, 8(1), 2004.

[5] X. Chen and T. Aamodt. A first-order fine-grained multithreaded throughput model. In *HPCA 2009: Proceedings of the 15th Annual IEEE International Symposium on High Performance Computing and Applications.*

[6] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *MICRO 2008: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture.*

[7] B. T. Davis. *Modern DRAM architectures.* PhD thesis, University of Michigan, 2001.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel® Technology Journal*, 5(1), 2001.

[9] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. To Appear In *ISCA 2009: Proceedings of the 36th Annual International Symposium on Computer Architecture.*

[10] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA 2004: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 338.

[11] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA 1981: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, 1981.

[12] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT 1999: Proceedings of the Eighth International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[13] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Int. J. Parallel Program.*, 29(1):35–58, 2001.

[14] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, 1994.

[15] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[16] NVIDIA Corporation. NVIDIA Quadro4 XGL. http://www.nvidia.com/page/quadro4xgl.html.

[17] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 1.1 edition, 2007.

[18] D. J. Ofelt. *Efficient Performance Prediction for Modern Microprocessors.* PhD thesis, Stanford University, 1999.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA 2000: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138.

[20] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. M. W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *CGO 2008: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204.

[21] Samsung. 512Mbit GDDR3 SDRAM, Revision 1.0 (Part No. K4J52324QC-B). http://www.alldatasheet.com/datasheet-pdf/pdf/112724/SAMSUNG/K4J52324QC.html, March 2005.

[22] J. G. Shanthikumar and R. G. Sargent. A Unifying View of Hybrid Simulation/Analytic Models and Modeling. *Operations Research*, pages 1030–1052, 1983.

[23] D. T. Wang. *Modern DRAM Memory Systems: Performance Analysis and a High Performance Power-Constrained DRAM Scheduling Algorithm.* PhD thesis, University of Maryland at College Park, 2005.