Parameterized DRAM Model

by Yong-jin Kwon
(Kwon@berkeley.edu)

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanovic

Research Advisor

(Date)

* * * * * * *

Professor XXX

Second Reader

(Date)

1. Introduction

Having just presented *Silicon-Photonic Clos Networks for Global On-Chip Communication* in the 3rd ACM/IEEE International Symposium on Networks-on-Chip conference, our research group found the usefulness of silicon photonics in core-to-core communications somewhat limited and wanted to explore a more viable use of monolithic silicon photonics. We determined that silicon photonics in DRAM architecture was a natural progression from our previous studies because the DRAM system included long-distance communication that photonics could easily improve.

The *Parameterized DRAM Model* (PDM) was initially a summer project intended to be used to explore the use of monolithic silicon photonics in the DRAM system. PDM is a robust DRAM simulator written using Chris Batten's cycle-accurate micro architectural C++ Hardware Simulator framework (HSIM) described in Appendix A. When PDM was first designed, we were not certain if we would completely rebuild the DRAM system architecture or modify an existing solution today. Because we wanted to explore a potentially large scope of designs, PDM was designed to maximize customizability and adaptability. Although this document is ultimately intended to show how to use PDM, it also outlines design decisions as well as missing features for anyone interested in developing future DRAM simulation frameworks.

In this report, we first discuss current DRAM technology as well as the general memory system architecture and we isolate an interesting set of design spaces. We show how PDM deals with these design spaces and show how to implement an existing DRAM interface using PDM. Lastly, we discuss what is not included in PDM currently and why they were omitted. PDM has been proven to be a very useful tool in designing and analyzing new DRAM architectures.

2. DRAM Technology

The fundamental blocks used in DRAM are the same regardless of which DRAM interface (DDR, SD, RAMBUS) is used. In this section, we outline the DRAM chip technology and various key terms as written in our ISCA publication (Beamer, et al., 2010). Figure 1 shows the structure of modern DRAMs which employ multiple levels of hierarchy to provide fast, energy-efficient access to billions of storage cells. At the lowest level, each *cell* contains a transistor and a capacitor and holds one bit of storage.

Cells are packed into 2D arrays and combined with the periphery circuitry to form an *array core* (Figure 1 (a)). Each row of the array core shares a wordline with peripheral wordline drivers, and each

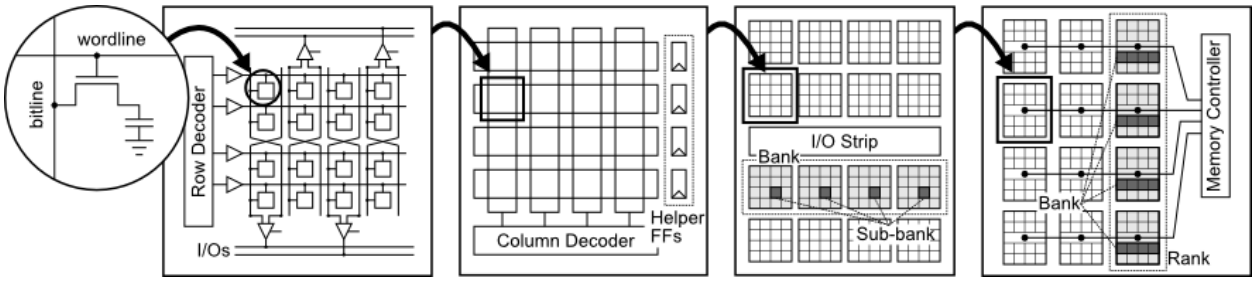


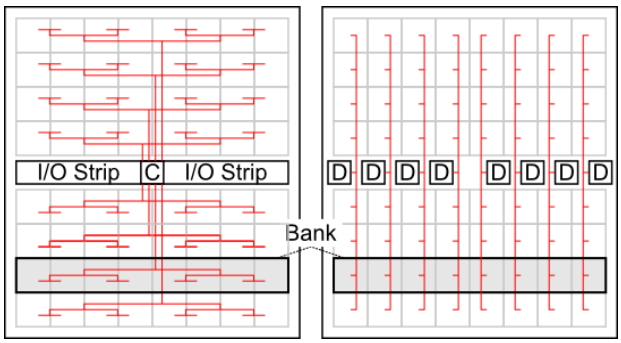
Figure 1: DRAM Memory System - Each inset shows detail for a different level of current electrical DRAM memory systems.

column shares a bitline with peripheral sense-amplifiers. Differential sense-amplifiers are used to amplify and latch low-swing signals when reading from the bitlines and to regenerate full-rail voltages to refresh the cell or write new values into the cell. The array core is sized for maximum cell density for a reasonable delay and energy per activation or refresh. Even though every cell in the activated row is read on an activation of an array core, only a few bits will be transferred over the array core I/O lines during a column access.

An *array block* is a group of array cores that share circuitry such that only one of the array cores is active at a time (Figure 1 (b)). Each array core shares its sense-amplifiers and I/O lines with the array cores physically located above and below it, and the array block provides its cores with a global predecoder and shared helper flip-flops for latching data signals entering or leaving the array block. As a result, the access width of an array block is equivalent to the number of I/O lines from a single array core.

A *bank* is an independently controllable unit that is made up of several array blocks working together in lockstep (Figure 1 (c)). The number of array blocks per bank sets the bank's access width. Array blocks from the same bank do not need to be placed near each other, and they are often striped across the chip to ease interfacing with the chip I/O pins. When a bank is accessed, all of its array blocks are activated, each of which activates one array core, each of which activates one row. The set of activated array cores within a bank is the *sub-bank* and the set of all activated rows is the *page*.

A *chip* includes multiple banks that share the chip's I/O pins to reduce overheads and help hide bank busy times (Figure 1 (c)). Figure 2 shows how the I/O strip for the off-chip pads and drivers connects to the array blocks in each bank. The DRAM command bus must be available to every array block in the chip, so a gated hierarchical H-tree bus is used to distribute control and address information from the centralized command pins in the middle of the I/O strip (Figure 2 (a)). The read- and write-data



(a) Command Bus (b) Read & Write Data Bus

Figure 2: DRAM Chip Organization – Example DRAM chip with eight banks and eight array blocks per bank: (a) command bus is often implemented with an H-tree to broadcast control bits from the command I/O pins to all array blocks on the chip, (b) the read- and write-data buses and array blocks are bit-sliced across the chip to match the data I/O pins. (C = off-chip command I/O pins, D = off-chip data I/O pins, on-chip electrical buses shown in red)

buses are striped across the chip such that all array blocks in a column are connected to the same data bus pin in the I/O strip (Figure 2 (b)).

A *channel* uses a memory controller to manage a collection of banks distributed across one or more DRAM chips (Figure 1 (d)). The channel includes three logical buses: the command bus, the read-data bus, and the write-data bus. To increase bandwidth, multiple DRAM chips are often ganged in parallel as a *rank*, with a slice of each bank present on each chip. To further scale bandwidth, the system can have multiple memory channels. To increase capacity, multiple ranks can be placed on the same channel, but with only one accessed at a time.

3. Memory System Architecture

A diagram of a modern day memory system is outlined in Figure 3. At the most abstract level, a memory system is three logical modules (CPU, memory controller and DRAM) connected with two logical interconnects. These modules and interconnects are described as 'logical' because they might be implemented with many different physical entities or combined as one. A memory request originates at the CPU (which includes an arbitrary number of caches) and travels to the memory controller using the request interconnect. Within the memory controller, a series of memory requests are converted to DRAM commands and DRAM write-data which are sent to the DRAM using the DRAM interconnect. The data produced by the DRAM module (DRAM read-data) take the reverse route through the DRAM interconnect to the memory controller where they are converted into memory responses which are directed to the CPU through the response network. Different DRAM interfaces are created by varying three basic components in the described memory system architecture. These basic variables can be

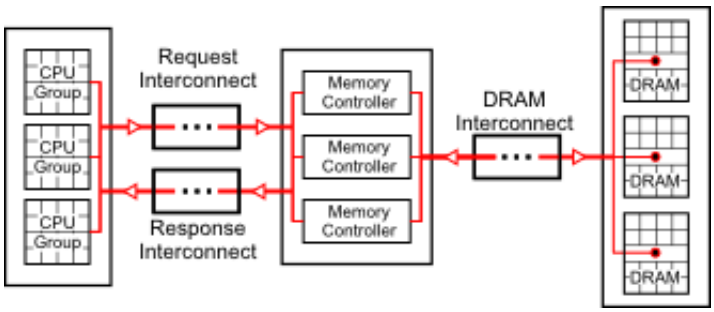


Figure 3: Top Level DRAM System Architecture – The three logical modules of a DRAM system (CPU, memory controller, and DRAM) and the two logical interconnects that connect them.

summarized by asking three questions: what is being transported, when is it being transported, and how is it being transported.

What is being transported? Within the three logical interconnects, we have a total of five different payload types found in the DRAM system: request, response, DRAM command, DRAM read data, and DRAM write data. Because the request and response packets are generally defined by the CPU, these payloads are assumed to be unchanged across different DRAM interfaces. The DRAM data is generally raw bits and does not require further customizability besides bit width. Therefore, the first interesting parameters are the DRAM commands because they define how the memory controller will interface with the DRAM.

When is it being transported? The second interesting parameter is the timing and delay associated with DRAM commands and data. Not only are the timing and delay widely different based on a specific DRAM part, but the same DRAM part can be initialized with different timing and delay parameters. In order for PDM to be able to support a wide array of DRAM interfaces, it must be able to deal with a wide variety of timing parameters necessary to run the DRAM interface correctly.

How is it being transported? The third interesting parameter determines the implementation of the two logical interconnects that bridge between the CPU, memory controller and DRAM. Not only can these interconnects be implemented with any physical topology, but two separate physical interconnect networks can be used to implement each logical interconnect. For example, the DRAM interconnect, which is simply a collection of channels, is commonly separated into command and data interconnects. However, this is not always the case since the original implementation of Rambus had command and data shared the same physical link.

To be robust enough to implement a wide variety of DRAM interfaces, the DRAM simulator needs to be able to easily deal with all three parameters.

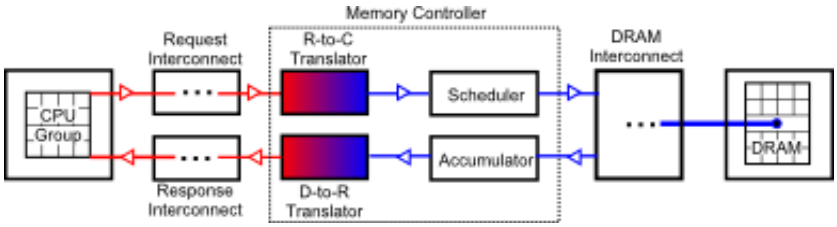


Figure 4: Simulator Block Diagram – This is a diagram of a single channel in the simulator. Notice how the simulator structure is essentially implements the Top Level Dram System Architecture shown in Figure 3. (R-to-C = Request to Command, D-to-R = Data to Response)

4. Building the Memory Simulator

Figure 4 shows a high level block diagram of PDM. The simulator was designed with modularized components to maximize sharing between implementations of DRAM interfaces. The reason PDM can be used to implement such a large design space is that each block is independent and does not assume a specific operation about any other blocks in the system. The following subsection discusses how PDM's modular architecture efficiently deals with the three parameters discussed in the previous section.

Making DRAM Commands Configurable

The ultimate vision for configurable DRAM commands is for a user to be able to reuse the same basic components across many different sets of DRAM commands. For example, let's assume that we want to add a new power saving command into an already existing DRAM interface. If we were to add the new command into an already existing implementation of the original DRAM interface, it should keep the behavior of the simulation unchanged.

For this reason, a standard enumeration cannot be used to implement polymorphic commands. Standard C++ enumerations have some problems including type-safety issues, lack of a containing namespace, inability to easily read/write enumerations to streams, and no support for extending on enumerations. In C++, enumerations are essentially names bound to integer values. Standard enums are incredibly difficult to pass between namespaces and finding all available enum values from an enum type is virtually impossible to do.

PDM uses a custom enumeration implementation called stdxEnum2 to bypass the problems caused by traditional enumeration types. A wrapper class (currently called DramCommand) instantiates a stdxEnum2 type that contains all the wanted commands as well as some helper methods used to

access other meta data such as address. All modules which deal with the DRAM command now have convenient access to a list of all possible commands through the wrapper class.

The ability to add new commands at will is a powerful tool for incremental development. We can first implement a small subset of available commands in a DRAM interface and incrementally add new commands to verify correct behavior of the added commands.

Making Timing Configurable

In a DRAM system, there are multiple locations that require checking the timing of commands. One location is the memory controller because it will need a way to verify correct timing to issue commands at the correct time. We also might want to place timing checks in the banks and ranks to make sure our memory controller is not violating any timing. Because PDM is designed to be modular, not all components are required to be written in C++. Any single block in Figure 4 can be replaced with a Verilog simulation without compromising the behavior of the simulation. Checking timing in the bank and rank is essential if your software simulator is able to replace individual modules with Verilog modules because we could easily verify any problems in timing caused by the RTL block.

Because we might want to verify the timing in multiple locations, PDM uses an easily modifiable verifier that can be instantiated anywhere. When the verifier is declared, the user passes in two template arguments that list all the commands and states the DRAM interface uses. Once the verifier is declared, the user passes in a series of transitions and constraints to configure the verifier. A *transition* is composed of a source state, a destination state, and a trigger command that allows the verifier to transition from the source to destination state. The transitions successfully define a state machine for the DRAM and allow the verifier to determine what commands are legal at any state. A *constraint* is composed of two commands and a cycle number that signifies how many cycles after the first command one needs to wait until the second command is valid. The constraint also contains two Boolean variables that define whether the timing constraint is placed in the same or different ranks and banks. By using both transitions and constraints, we can successfully define when and if a command can be issued. By changing what transitions and constraints are entered into the verifier, we can define a very large array of DRAM interfaces.

Once all the transitions and constraints have been correctly set in the verifier, the verifier can be exercised with two methods: check and set. The check method is called when we want to check if a specific command is valid. For example, we can use the check method in the memory controller to poll

if we can issue the next command in the queue. Another use is to place a verifier in the bank or rank and check to see if the incoming command meets the timing requirements. The set method is used once we verify that the command is valid and we either issued it or accepted the command.

Making Interconnects Configurable

Making the interconnect in the DRAM implementation configurable does not require much in PDM. Since all the modules in PDM use the same handshaking interface, we can place any kind of interconnect between the modules and guarantee correct behavior. One obvious caveat is that if the packet transported is time sensitive (such as a DRAM command), the interconnect will require a

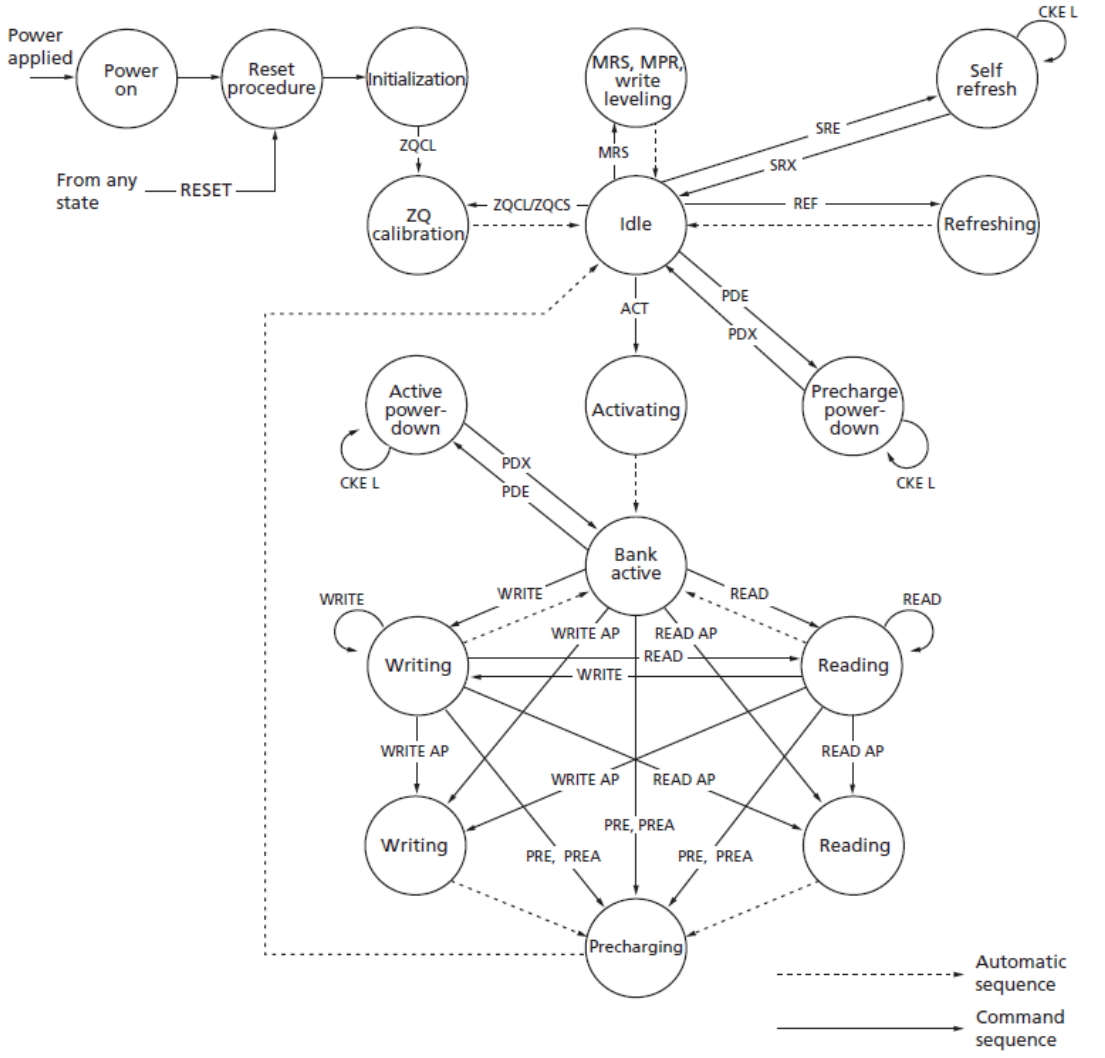


Figure 5: DDR3 State Machine – State machine found in the Micron DDR3 datasheet (Micron)

predictable amount of latency for the source to time the packet correctly. This means we generally do not want interconnects that introduce variance in latency due to congestion and routing.

5. Configuring the Simulator

In this section, we describe how a specific DRAM interface is implemented in PDM. Because it is commonly used in modern day systems, the DDR3 interface is outlined in the following subsection and implemented in PDM.

The DDR3 Interface

Figure 5 outlines a simple state machine for the DDR3 interface. PDM does not simulate the initialization of the DRAM. Initialization of the DRAM is used to configure certain parameters such as timing and command behavior and is not interesting for analysis because it is always constant and predictable. Because we are using a C++ simulation, parameters can be passed into the DRAM module upon instantiation.

Assuming the DRAM is already initialized, there are six core commands that the DRAM can recognize: activate, read, write, precharge, refresh, and power down. These DRAM commands use the command bus to broadcast to all the banks and ranks and therefore most commands require a few bits to specify which rank and bank the command is intended for. Some of the power down commands utilizes this broadcasting ability to target a group of banks to power down.

The *activate* command takes a row number and opens a row in a particular bank for a subsequent access. Opening a row essentially stores an entire row of data into the sense amps such that a later access can be used to retrieve a small chunk of the preloaded row. This row remains open for further accesses until a precharge command is issued to that bank.

A *read* command is used to initiate a burst read access to an active row while a *write* command is used for a burst write access. A burst is defined as either four or eight columns depending on a status bit and cannot be interrupted once the command is issued. Both read and write commands require the use of the data bus. The data bus, like the command bus, is shared across all banks and ranks. However, the data bus is bidirectional while the command bus is unidirectional. With a read command, a burst of data is placed on the data bus a set number of cycles after the read command and conversely a burst of data is required on the data bus a set number of cycles after a write command. The timing of DRAM

commands must be very precise for the data bus to be utilized by only a single bank at any given time. Any read or write commands can be instructed to automatically precharge after the operation is complete. If auto precharge is not selected, the opened row will remain open after the read/write operation. When the auto precharge is selected this is referred as a closed read or write as opposed to an open read or write.

A *precharge* command is used to deactivate the open row in a particular bank or in all banks. During a precharge, the bit lines are recharged such that a new row can be loaded with an activate command. During this precharge period, all reads and writes to other banks can be successfully issued without disturbing the precharge. After the precharge period, the bank is returned to the idle state where it waits for another activate command.

Since DDR3 is a dynamic memory, a cell requires regular refreshing for the data to remain uncorrupted. For DDR3 this refresh process is invoked by the refresh command. A *refresh* can be targeted to one bank or all banks and no addresses are needed because the addresses are computed using an internal counter.

The *power down* command can be issued to shut down the DRAM device to save power. If the DRAM device is not to be used for a long duration, the memory controller can decide to issue the power down command.

PDM Implementation of DDR3

This section shows how PDM is set up for the DDR3 DRAM interface. Using PDM we can easily create an implementation of any other DRAM interface using similar techniques.

CPU

In PDM, the CPU is emulated with a combined source and sink module which injects preset traffic patterns into the memory system. Two different groups of traffic patterns are currently implemented in PDM: random and streaming. The random traffic pattern generates a random base address and uses an offset of the base address for a given number of accesses. After a “stream size” number of memory accesses, the random traffic pattern generates a new random base address and the process continues. If the stream size is set to one, the traffic pattern would be a true random traffic pattern where the address is randomized for each access. The request is randomly chosen to be a load or store based on a parameterized load to store ratio.

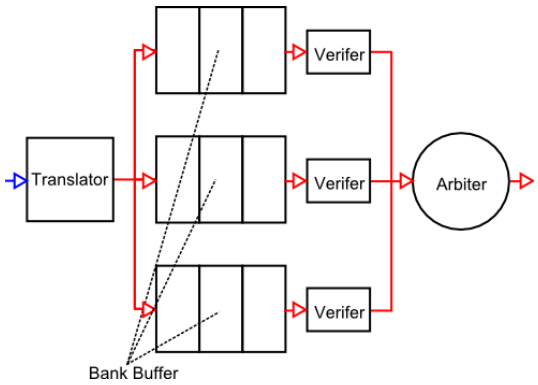


Figure 6: Memory Controller – High level block diagram of the KDM DDR3 memory controller implementation

With the streaming traffic pattern, the user defines either a traditional triad or mem copy operation. In mem copy, the pattern defines two streams: one to read and one to write, a common operation when copying a long vector. Triad is a stream operation which defines two read streams and one write stream, simulating an arithmetic operation on two vectors and storing to a third.

Since a synthetic traffic pattern is generally representative of a single core operation, we can create a simulated multicore traffic pattern by connecting many sources with a crossbar. PDM allows an arbitrary number of sources to be bundled to represent any number of threads being run at the same time. A creative mix of different traffic patterns can be grouped easily to generate new traffic patterns as needed.

DRAM

The PDMDramBank module simulates a DDR3 DRAM part by instantiating a 2D array of bytes the size of the DRAM component. PDM instantiates this large array to make sure the correct data is being written and read back. Not only does this allow for comprehensive tests, but if PDM were to eventually support any RTL model, storing and reading the correct data will be a valuable resource for verifying correct behavior.

Memory Controller

A high level block diagram of the memory controller (pdmDramMemoryController) is shown in Figure 6. The memory controller can be divided into three units: the translator, the arbiter, and the scheduler (verifier). The translator takes the memory requests and converts them into an optimal sequence of DRAM commands. These commands are then placed in buffers based on the intended bank. The verifier checks whether the bank buffers with commands are able to be issued on the next cycle. The arbiter then uses some arbitration scheme to determine which bank buffers to issue. The

arbitration scheme currently used in PDM for DDR3 is round robin among the banks that have been filtered by the verifier. In this case, PDMDramSimpleDDRVerifier implements a simple DDR3 verifier using timing parameters in Table 1.

Table 1: DDR3 Timing Constraints – The list of DDR3 Constraints (as defined in Section 4). The tCONSTANT values can be extracted from the DDR3 data sheet (Micron).

Last Command	Next Command	Rank	Bank	Timing
Activate	Activate	Same	Same	tRC
Activate	Activate	Same	Different	tRRD
Precharge	Activate	Same	Same	tRP
Refresh	Activate	Same	Same	tRFC
Activate	Read	Same	Same	tRCD-tAL
Read	Read	Same	Any	tCCD
Read	Read	Different	Any	tCCD-tRTRS
Write	Read	Any	Any	tCCD-tWTR
Activate	Write	Same	Same	tRCD-tAL
Read	Write	Any	Any	tCAS+tCCD+tRTRS-tCWD
Write	Write	Any	Any	tCCD
Activate	Precharge	Same	Same	tRAS
Read	Precharge	Same	Same	tRTP+tAL
Write	Precharge	Same	Same	tCWD+tCCD+tWR
Refresh	Refresh	Same	Any	tRFC
Precharge	Refresh	Same	Any	tRP

Interconnect

The DRAM interconnect described previously is simply a collection of channels. A channel is implemented as a collection of buses in a DDR3 system. PDM uses two different types of buses to create this DRAM interconnect: a unidirectional bus and a bidirectional bus. The unidirectional bus is used to transport the necessary commands from the memory controller to the appropriate bank while the bidirectional bus is used for data. Both buses are designed such that only one input is allowed access at the same time.

6. Future Work

PDM's representation of the DDR3 DRAM interface is not complete. In this section we outline some missing components that are required for a more robust DRAM interface implementation. The verifier used in the DDR3 implementation only implements the most commonly used commands. As of yet, PDM's implementation of DDR3 lacks the ability to do broadcast commands as well as any type of closed read/write commands because using such commands require a smarter memory controller. PDM's memory controller implementation is still very basic and not customizable. For instance, the arbitration scheme is not modular and only round robin is currently possible. For PDM to implement an ever larger array of DRAM interfaces, the memory controller will need to be reworked to allow more customization and extensions.

The bus implementations in PDM also need to be improved. Currently, the entire PDM DRAM system shares the same clock and there is no way to fine tune the bus speeds. The only way to accomplish this is to access a larger chunk of data per cycle to simulate a faster bus speed. However, this is not always easy to accomplish if the desired bus speedup is not a whole number. Also, the command bus bandwidth is currently difficult to scale up because of this limitation. If we were to double the command bandwidth, we would have to instantiate two command buses for our implementation to work.

7. Results

As the simulation runs, PDM keeps track of various events such as latency and outputs the aggregated results once the simulation completes. Dividing the number of cycles our data bus carries data by the total simulation cycle count gives us the data bus utilization rate. The data bus utilization rate of a large number of configurations is graphed in Figure 7. In Figure 7 (a)(b)(c), we vary the load to store ratio (or "loadRate") with a stream size of one in our random traffic generator. Figure 7 (d)(e)(f) plots the same but with a stream size of four. Within each graph, we graph a series of data utilization versus the number of banks per channel for varying number of capped memory requests in flight.

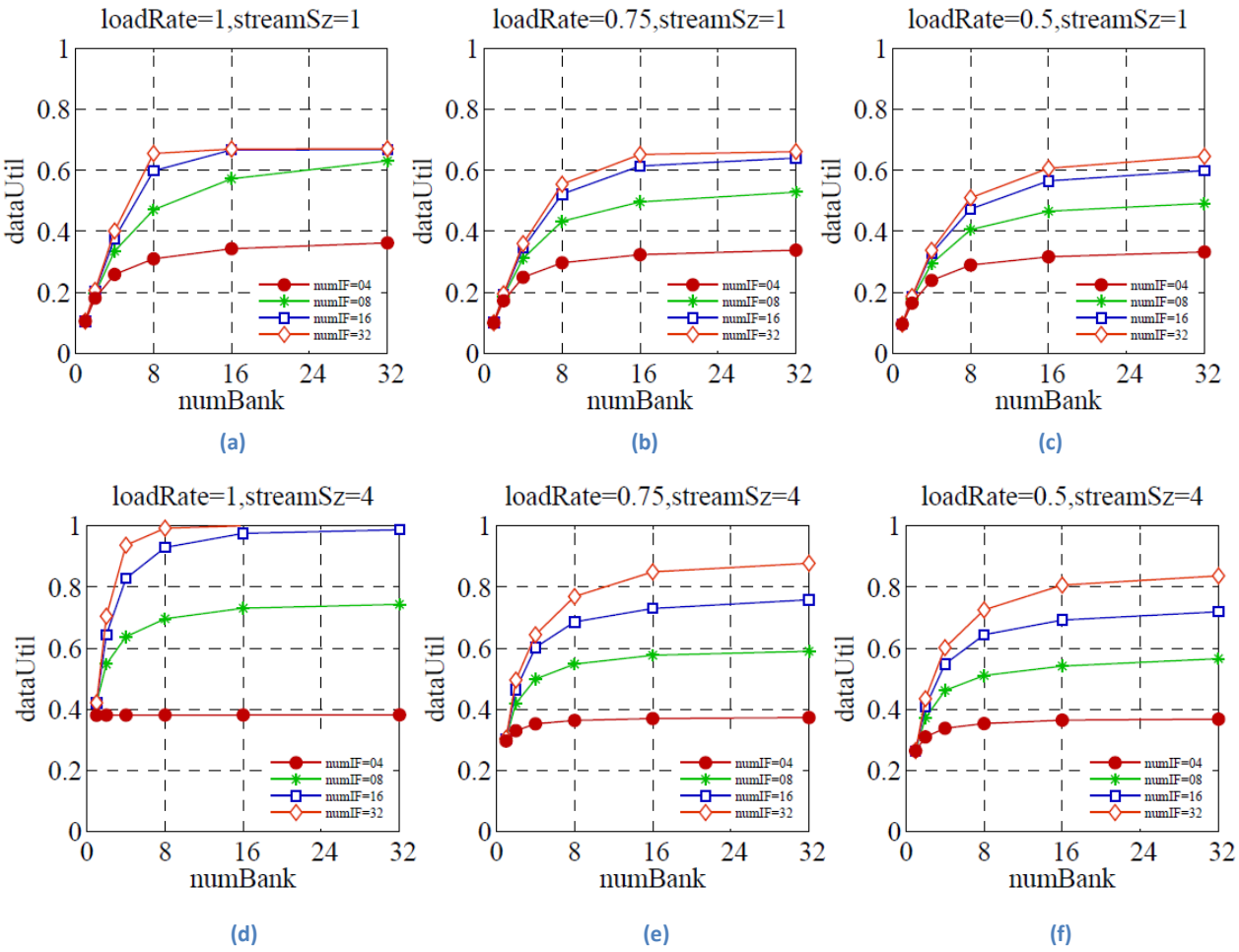


Figure 7: Example Event Data 2 – (DataUtil = data utilization rate, numBank = number of banks per channel, loadRate = load-to-store ratio, StreamSz = size of the traffic pattern stream, numIF = number of in-flight messages)

There are two interesting observations we can make from these results. Taking a look at Figure 7 (a) and Figure 7 (d), we can see that our memory controller is unable to utilize the data bus at full capacity with a stream size of one because our timing parameters place limits on consecutive activates to different banks (refer to Table 1). We are able to fully saturate the data bus with four streams because each activate command is timed four read burst apart, giving the next activate command enough time to meet the activate–to-activate timing constraint.

Another interesting observation is the dip in data bus utilization as we vary the load-to-store ratio. When we only issue loads (load rate is 0), we see maximum data bus utilization because there are no timing constraints placed between consecutive loads. However, as we get an even mix of loads and stores, we start to experience the read to write and write to read timing constraints eventually driving data bus utilization down.

8. Conclusion

The data in the evaluation section of *Re-Architecting DRAM Memory Systems with Monolitically Integrated Silicon Photonics* was generated using a combination of PDM and a heavily modified version of the CACTI-D DRAM modeling tool. A series of events generated from PDM was used in conjunction with the modified CACTI-D tool to generate energy and power numbers for analysis. Although PDM is not complete, it a tool robust enough to explore new DRAM architectures. With small modifications to the memory controllers to allow more customization, PDM can become a very powerful tool for exploring DRAM system architectures.

9. Bibliography

Beamer, S. et al. (2010). Re-Architecting DRAM Memory Systems with Monolitically Integrated Silicon Photonics. *ISCA.*

Micron. *Micron DDR SDRAM products. Online Datasheet,.* Retrieved from http://www.micron.com/products/dram/ddr3.

Appendix A: HSIM

PDM is implemented as a project in Chris Batten's Hardware Simulator (HSIM). This appendix outlines the basic usage of HSIM by showing how to configure and run PDM. This is not intended to be a full tutorial on using HSIM but a quick guide to getting started.

Folder Hierarchy

The top level directory of HSIM contains two directories named build and sims.

Sims Directory

The sims directory is where all the projects and their relevant source code is located. Taking a look into pdm project shows a series of header and source files as well as a module.mk file used to build the project. File extensions used are universal throughout the project where *.u.cc corresponds to a unit test of the *.cc/h file and the *.t.cc is the templated member function definitions of the *.h file.

The module.mk file in each project is required to build the project. The module_deps variable lists which project in the sims directory this project depends on. The hdrs, srcs, tmpl_srcs, and utst_srcs each corresponds to the header, source, template, and unit test files for each module in the PDM

project. Finally, the prog_srcs variable defines all non-unit-test project files that are executable. Notice the only prog_srcs file is pdm-system-numbanks.cc.

Build Directory

The build directory is used to compile and run all unit tests and project executables. Any scripts and configuration files used for the project is also contained in the build directory. All compilations are done in the gcc-comp-sims directory. To compile, we run the configuration script in the sims directory from gcc-comp-sims via '../../sims/configure' and 'make.' Several useful make arguments are available: run-all-tests (compile and run all unit tests) and *-unit-test (compiles a specific unit test which can be run via ./*-unit-test). To compile PDM and all dependent projects and files, invoke 'make pdm-system-numbanks.'

Configuration and Running PDM

Once PDM has been successfully compiled, we can simply run ./pdm-system-numbanks with appropriate command line variables to set parameters. However, this is a very cumbersome process that can be simplified using configuration files and scripts to run the simulation. A set of example scripts can be found in the build directory under isca10 (the project file for the ISCA 2010 submission from our research group). There are two important scripts in the isca 2010 directory. One is the gen-cfg.pl perl script which generates a series of configuration files and the Makefile which runs many instance of the project with the configuration files from the perl script.

The Makefile has been set up such that invoking the command 'make cfgs' will run the perl script to generate configuration files. Once the configuration files have been created, we can then run 'make' to run all the tests. All outputs will be saved as .out and .event files once we run the project. The project has also been set up to display useful traces with the command –log-level=moderate.