DRAMSim2: A Cycle Accurate Memory System Simulator

Paul Rosenfeld, Elliott Cooper-Balis, Bruce Jacob
Department of Electrical and Computer Engineering
University of Maryland College Park
{prosenf1,ecc17,blj}@umd.edu

***Abstract*—In this paper we present DRAMSim2, a cycle accurate memory system simulator. The goal of DRAMSim2 is to be an accurate and publicly available DDR2/3 memory system model which can be used in both full system and trace-based simulations. We describe the process of validating DRAMSim2 timing against manufacturer Verilog models in an effort to prove the accuracy of simulation results. We outline the combination of DRAMSim2 with a cycle-accurate x86 simulator that can be used to perform full system simulations. Finally, we discuss DRAMVis, a visualization tool that can be used to graph and compare the results of DRAMSim2 simulations.**

I. Introduction

Simulation has become a vital tool for researchers in the computer architecture community. Evidence of this can be seen in the proliferation of academic and commercial simulators of different architectures, levels of detail, and features. Unfortunately, many of these CPU simulators overlook the need for accurate models of the memory system. Many such simulators include simplistic models of memory which fail to account for the highly complex behavior of modern memory systems. A typical DDR memory controller reorders and schedules requests multiple times while keeping track of dozens of timing parameters. Despite the fact that main memory is growing both in complexity and as a system-level bottleneck, 'cycle accurate' CPU simulators often attach a fixed latency to memory accesses, significantly under-reporting the real effect of the memory system.

To help fill the void of accurate memory system simulators, we present DRAMSim2[1], a cycle-accurate DDR2/3 simulator. The major goal of DRAMSim2 is to provide an easy to integrate and accurate DDR2/3 memory model. DRAMSim2 focuses on having a very simple programming interface and object oriented design while still being configurable. Additionally, DRAMSim2 provides a verification tool that can be used to validate the results of any DRAMSim2 simulation, regardless of the front end driver[2] Finally, DRAMSim2 provides a robust visualization tool, DRAMVis, that enables the user to visualize and compare the effects of memory system parameters on key performance metrics such as bandwidth, latency, and power. A focus on usability, ease of integration, and supporting tools is what motivated us to create a brand new simulator instead of using an existing one. We believe these features make DRAMSim2 a valuable tool for computer architecture researchers that want to include memory models in their research.

One of the first attempts to provide a detailed model of main memory was DRAMSim [1], software designed to explore memory systems concepts by providing models for many types of memory systems (such as SDRAM, DDR, and FBDIMM). Several other DRAM simulators have been mentioned in the literature such as dsim [2] and rascas [3], but neither simulator appears to be publicly available today.

[1]DRAMSim2 is available at http://www.ece.umd.edu/dramsim/

[2]A "driver" is any code that instantiates a `MemorySystem` object and issues requests (e.g. trace, CPU simulator, etc.)

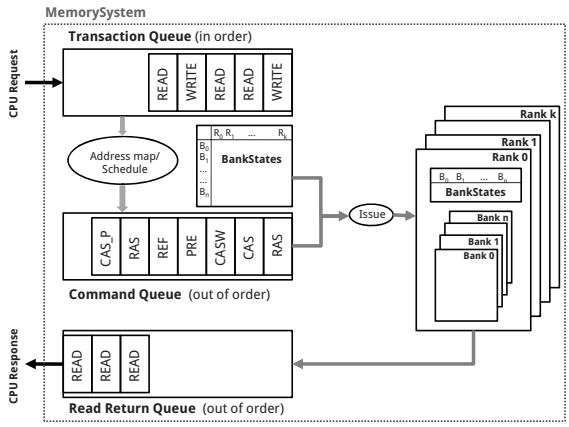

Fig. 1. Overview of DRAMSim2 components. The `MemorySystem` wrapper provides a simple user interface for any front end driver.

The structure of the paper is as follows: section II describes the basic architecture of DRAMSim2, section III talks about the challenges and approach to verifying the simulator, section IV presents a full system simulation environment using the MARSSx86 simulator, section V describes DRAMVis, our cross platform visualization tool for DRAMSim2, and VI concludes.

II. DRAMSim2 Architecture

DRAMSim2 is implemented in C++ as an object oriented model of a DDR2/3 memory system that includes a detailed, cycle accurate model of a memory controller that issues commands to a set of DRAM devices attached to a standard memory bus. A block diagram of the major components of DRAMSim2 is illustrated in Fig.1. The DRAMSim2 core is wrapped in a single object called `MemorySystem`. A memory system object requires two ini files: a device ini file and a system ini file. A device ini file contains parameters that describe a specific DRAM device such as the timing constraints and power consumption of the device. These parameters can be found on manufacturer data sheets which are available on their websites. The DRAMSim2 package contains several device ini files for Micron DDR2/3 devices of varying densities and speed grades. The system ini file consists of parameters that are independent of the actual DRAM device. These include parameters such as the number of ranks, the address mapping scheme, debug options, row buffer policies, memory controller queue structures, and other simulation details.

After a `MemorySystem` object is created, the driver code registers a callback function to be executed when a request completes. At this point, the initialization of DRAMSim2 is complete. The driver code must call a function for each memory system clock tick and another function to add memory requests. Upon completion of a memory

Posted to the IEEE & CSDL on 3/11/2011
DOI 10.1109/L-CA.2011.4 1556-6056/11/$26.00 © 2011 Published by the IEEE Computer Society

request several clock cycles later, DRAMSim2 calls the supplied callback function to let the front end driver know that the request is finished. The number of clock cycles between when a request is added to DRAMSim2 and when it completes varies with the current memory system state. Because the internals of the simulator are encapsulated into an interface that consists of just a few functions, it is easy to attach DRAMSim2 to any type of front end driver such as a trace reader, a cycle accurate CPU simulator such as MARSSx86 [4], or a discrete event simulation framework such as SST [5].

DRAMSim2 can be compiled either as a standalone binary or as a shared library. In standalone mode, DRAMSim2 simulates commands read from a memory trace on disk. In the shared library mode, DRAMSim2 exposes the basic functionality to create a `MemorySystem` object and add requests to it.

Other than the C++ STL, DRAMSim2 relies on no external libraries; thus, it is easy to build on any platform that has the GNU C++ compiler installed. We have successfully built and run DRAMSim2 on Linux, Windows (with Cygwin), and OSX.

A. Simulation Internals

Because manufacturers avoid publishing details about the internals of their memory controllers, DRAMSim2 attempts to model a modern DDR2/3 memory controller in a general way [6]. Requests from the CPU are buffered into a *transaction queue* in execution order. These transactions are converted into DRAM commands and placed into a *command queue* which can have different structures such as *per rank* or *per rank per bank*. The memory controller maintains the state of every memory bank in the system and uses this information to decide which request should be issued next. The memory controller is free to issue requests from the command queue out of order as long as it doesn't schedule writes ahead of dependent reads or violate timing constraints. Issuing requests out of order helps to increase bank usage and therefore helps increase bandwidth and lower latency. In *open page* mode, DRAMSim2 will keep rows open and prioritize requests to already open rows. This can decrease the overhead of precharging rows that might potentially be used in the near future. Alternatively, *closed page* mode will precharge rows immediately after each request; this can be beneficial for workloads with poor locality.

After the simulated DRAM devices receive the commands and data from the memory controller, a second set of bank states at the ranks are used for error checking to make sure that the timing of the received command is valid. If the simulator is compiled in a timing-only mode, data from writes are not stored, and data is not included in read responses. Alternatively, DRAMSim2 can be configured to store data on writes and return data on reads. Many CPU simulators and other front-end drivers are not concerned with the actual data, so data storage is disabled by default to reduce the memory usage of the simulator.

As reads and writes complete, the simulator keeps track of the bandwidth and latency of the requests. These statistics are averaged over an *epoch*, the length of which is configurable by the user. During each epoch, the simulator outputs detailed bandwidth, latency, and power statistics both to a .vis file and the console (or a log file). The resulting .vis file can be loaded into DRAMVis, which is described in section V.

In addition to simulating the state changes due to reads and writes from the driver, the DRAMSim2 memory controller also models the effects of DRAM refresh. Modeling refresh is important since refresh has become a major source of variance in the latency of memory requests. Read requests that are issued while a refresh is in progress have to wait much longer than other requests. This long latency can introduce major performance penalties for processors blocked on memory accesses and could significantly impact the performance of the system as a whole.

Finally, DRAMSim2 uses the power model described by Micron in [7] to compute the power consumption given the state transitions of each bank. DRAMSim2 also includes heuristics to place devices in low power mode to reduce power consumption during periods of low memory activity. The low power mode can be enabled or disabled with a flag in the ini file.

We omit the description of the actual DRAM timing parameters and their interactions since they are beyond the scope of this technical overview. A good description of these topics can be found in [1], [8], and [9].

III. Validation

A. Validation Challenges

One of the major design goals of DRAMSim2 has been to focus on trying to show that the simulation results are accurate. We believe that if one takes the time to create a memory model and the CPU time to run the model, then one should also make an effort to show that the model produces valid results. A memory system model is non-trivial to debug and validate; the memory controller's scheduling algorithm must take into account the state of every memory bank, its adherence to over a dozen timing parameters, the interactions between requests in multiple queues, and the arbitration of the bus that the devices share.

One of the most important guarantees that a memory simulator should make is that it does not violate the timing constraints of the DRAM device. Since these timing parameters represent the physical constants associated with a device, the simulator must take extra care to never issue commands faster than these parameters allow. Doing so would create unrealistic performance results with respect to actual hardware. This is the case that our validation process tries to eliminate since it is the most egregious type of error.

B. Verilog Model Verification

DRAM Manufacturers such as Micron often supply Verilog timing models for their DDR2/3 DRAM parts. Although default values are provided with the models, the Verilog timing parameters can be arbitrarily set, allowing the user to simulate any DDR2/3 device. The supplied files provide a set of Verilog tasks that simulate the toggling of command lines and the transmission of data to the DRAM devices.

DRAMSim2 models memory ranks receiving commands from a simulated memory controller. As DRAM commands (e.g., READ, WRITE, PRE) arrive at the simulated ranks, they can be captured along with their timing information to a log file. This file is processed by a small Ruby script to generate a Verilog output file containing calls to the corresponding Verilog task for that command. In other words, as the DRAMSim2 memory controller issues commands, these commands are executed by the Micron Verilog model with the same timing parameters; this indicates whether the DRAMSim2 memory controller has violated any timing constraints.

Three steps are required to run the verification suite. First, the `VERIFICATION_OUTPUT` option must be set in the system.ini file to tell DRAMSim2 to capture verification output. The relevant timing parameters from the device.ini file must be set in the Verilog parameters file by a script. After running DRAMSim2, the verification output must be post-processed to generate a Verilog test file. Finally, the parameters file and the test file are included along with the Micron Verilog files and executed by the ModelSim simulator. We

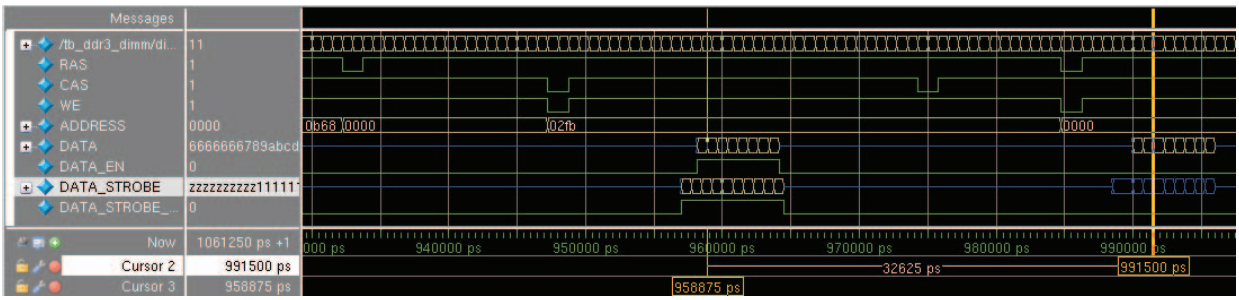

Fig. 2. ModelSim waveform output after running the verification script. Row 0 is activated in bank 0 and a burst of data is written to column 0x2fb. Then, without closing the row, column 0x2fb is read back and the data is compared to see if it is correct. Finally, a precharge command is issued to close the row.

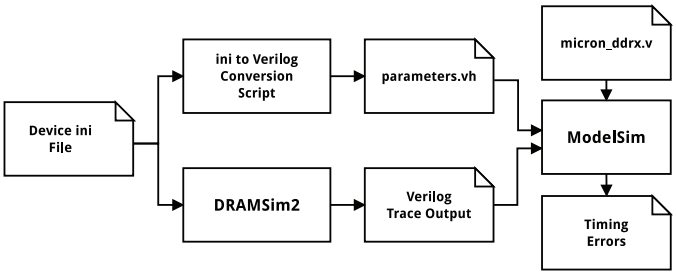

Fig. 3. An overview of the DRAMSim2 verification process.

have provided a small Ruby script that automates all of these tasks. The overall verification process is shown in Fig.3.

Currently, we have implemented two verification modes. In the timing-only mode, only a single device is instantiated per memory rank. Since all devices in a rank receive the same control signals, simulating one device per rank is sufficient to capture all timing violations. This mode executes quickly because the Verilog simulator must only model a single device per rank instead of the usual 4-16 devices per rank. The disadvantage of this mode is that only a portion of the data is retrieved on a read; thus, data verification is not possible.

The second verification mode populates a full rank of devices. The verification script generates `read_verify()` calls which contain the DRAMSim2 data that is read. The `read_verify()` command compares the DRAMSim2 data and the Verilog data to detect any data discrepancies as well as any timing violations. Unlike traditional unit tests that are carefully tailored to test specific cases, our validation procedure can be used with any DRAMSim2 simulation regardless of which driver generated the results. The Verilog simulations can be easily batched using ModelSim's command line interface or can be run interactively to generate a waveform output of the device signals such as in Fig.2. Although this type of validation scheme has been proposed in the past, to our knowledge it has never been implemented [3].

IV. Full System Simulation

Although memory is becoming more and more of a bottleneck, most CPU simulators have an extremely simple main memory model [10]. At best this consists of a bank conflict model: if subsequent requests go to the same memory bank, an extra fixed latency is added to the memory request. In the worst case, the memory model is a simple fixed latency. Even extremely detailed CPU simulators such as PTLSim [11] use a fixed latency model.

This approach is problematic because the dynamics of the CPU and memory are highly intertwined. Growing scheduling complexity in the memory system can result in highly variable latencies for memory requests based on the application request stream [8], [12]. Since processors spend a significant portion of execution time waiting for outstanding memory requests, accurately modeling the memory system is the only way to obtain meaningful full system results. Consequently, the real value of DRAMSim2 is to fill the void of publicly available DRAM simulators that are easily integrated into a full system simulation environment.

We set out to create such a simulation environment by choosing a CPU simulator and integrating DRAMSim2. We had several criteria in mind: multi-threaded simulation capability to run modern workloads capable of stressing the memory system; full system simulation mode to capture the effects of virtual memory and the kernel; reasonably fast simulation speed; an x86 CPU model; and the ability to integrate DRAMSim2 with minimal effort.

In the end, we chose a simulator called MARSSx86. This simulator expands on the extremely detailed x86 core models found in PTLSim/X by adding multicore support and a malleable MESI cache hierarchy. Additionally, MARSSx86 replaces the Xen hypervisor in PTLSim/X with the open source QEMU emulator, making it easier to integrate DRAMSim2. Although MARSSx86 runs completely in user mode, it achieves impressive simulation rates that have allowed us to run simulations with up to eight cores in a reasonable amount of time.

Integrating DRAMSim2 into MARSSx86 was relatively straight forward because both simulators are cycle-based. The MARSSx86 `MemoryHierarchy` object receives a clock tick of a user-controllable frequency. This clock is divided down and is used to call the DRAMSim2 clock function at the desired memory bus speed for the DRAM part. The default fixed memory latency in MARSSx86 is replaced with calls to add requests to DRAMSim2; a callback function within MARSSx86 sends these requests back through the cache hierarchy to the CPU when they are complete.

We simulated a dual-core CPU with 2GB of memory running a subset of the PARSEC [13] benchmarks. When compared with MARSSx86's default fixed latency memory model, the simulation time increased by between 16-53% (average of 30%) when using DRAMSim2. Though these benchmarks provide a rough estimate, the performance overheads of DRAMSim2 will vary greatly with system parameters such as number of cores, number of ranks, aggressiveness of scheduling policy, workload memory intensity, etc. However, we believe that the modest increase in simulation time is a fair price to pay for increased accuracy.

The patched MARSSx86 code that supports DRAMSim2 is publicly available through the DRAMSim2 website.

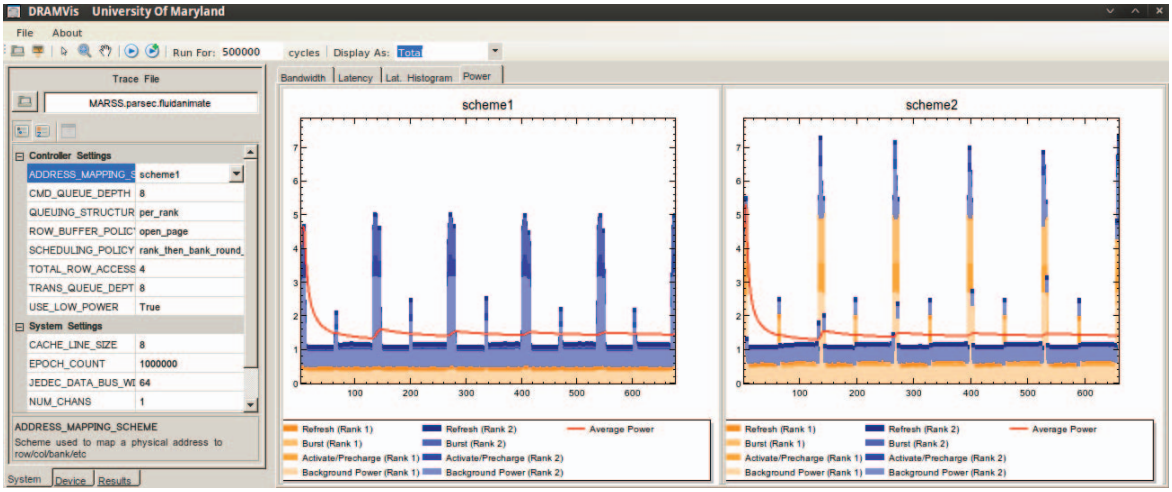

Fig. 4. DRAMVis comparing the effect of address mapping scheme on the power consumption of the "fluidanimate" benchmark from the PARSEC suite. Plots can be directly exported from the tool in png format.

V. Visualizing and Comparing Results: DRAMVis

By default, DRAMSim2 prints a summary of the bandwidth, latency, and power for each simulation epoch. While this approach can be useful for looking at the output of short simulations, it is not very enlightening for simulations of real programs that execute for billions of cycles. To help understand and compare the output of DRAMSim2, we've developed a visualization tool which we call DRAMVis. Every DRAMSim2 simulation generates a .vis file in the results directory. The .vis file contains a summary of the simulation parameters along with the power and performance statistics for each epoch. DRAMVis plots the .vis file data and displays the parameters of the simulation in the left pane. DRAMVis loads previous simulation results that can be plotted on the same axes for easy comparison.

Fig.4 shows the power consumption of a DRAMSim2 simulation of the PARSEC benchmark "fluidanimate". The simulation was run with MARSSx86 with two different address mapping schemes. The stacked bar graph shows the power components color coded by rank along with an average power line. While both graphs exhibit the same overall pattern, the right graph has higher power peaks and stresses both ranks of memory rather than just one. Furthermore, the higher power consumption is in part due to higher burst power, which implies higher bandwidth for the right graph (switching to the bandwidth tab and comparing the two graphs also confirms this observation).

DRAMVis supports standard graph interactions such as zooming and panning as well as breaking down latency and bandwidth by rank or by bank. Additionally, it can serve as a front end to run trace-based simulations. By changing some parameters and pressing the run button, DRAMVis invokes DRAMSim2 and adds the result to the interface, allowing the user to see instantly the effects of the parameter changes.

VI. Conclusion

In this paper we present a technical overview of DRAMSim2, a cycle-accurate memory system simulator. DRAMSim2 has a strong focus on being accurate and easy to integrate. We describe the simulator and our verification strategy as well as our full system simulation environment using MARSSx86. Additionally, we discuss our visualization and comparison tool, DRAMVis. The source code to all of the tools described in this paper are available on the DRAMSim2 website. Overall, we feel that DRAMSim2 is an invaluable tool for the growing use of simulation in the computer architecture field.

References

[1] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, 2005.

[2] S. Rixner, "Memory controller optimizations for web servers," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2004, pp. 355–366.

[3] J. Alakarhu and J. Niittylahti, "DRAM simulator for design and analysis of digital systems," *Microprocessors and Microsystems*, vol. 26, no. 4, pp. 189–198, 2002.

[4] A. Patel and F. Afram, "MARSSx86: Micro-ARchitectural and System Simulator for x86-based Systems," 2010. [Online]. Available: http://www.marss86.org/

[5] A. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, "The Structural Simulation Toolkit," 2010.

[6] N. Aboulenein, R. B. Osborne, R. Huggahalli, V. K. Madavarapu, and K. M. Crocker, "Method and apparatus for memory access scheduling to reduce memory access latency," sep 27 2001, US Patent App. 09/966,957.

[7] "TN-46-03 Calculating Memory System Power for DDR," Micron Technology, Tech. Rep., 2001.

[8] V. Cuppu and B. Jacob, "Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?" in *Proc. 28th Annual International Symposium on Computer Architecture (ISCA'01)*, vol. 29, no. 2. Goteborg, Sweden: ACM, jun 2001, pp. 62–71.

[9] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Pub, 2007.

[10] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer, "CMP Memory Modeling: How Much Does Accuracy Matter?" *Proc. Fifth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pp. 24–33, 2005.

[11] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *IEEE International Symposium on Performance Analysis of Systems & Software, 2007. ISPASS 2007*, april 2007, pp. 23–34.

[12] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A Performance Comparison of Contemporary DRAM Architectures," in *Proc. 26th Annual International Symposium on Computer Architecture (ISCA'99)*. Atlanta GA: Published by the IEEE Computer Society, may 1999, pp. 222–233.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.