P&S Ramulator
Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator

Haocong Luo
Prof. Onur Mutlu
ETH Zürich
Spring 2022
18 March 2022
A Tutorial on Understanding, Using, and Extending Ramulator
Understanding Ramulator
Simplified DRAM State Machine (Bank)
Responsibilities of the Memory Controller

- Interface between the processor and the DRAM device.
  - Queueing read and write requests from the processor
  - Translating the requests into actual DRAM commands (e.g., PRE, ACT, RD/WR).

- Schedule the requests while obeying the timing constraints.
  - Pick a request according to the scheduling policy and issue the corresponding DRAM commands (according to the current state of the DRAM device).
  - Timing constraints between DRAM commands must be obeyed.

- Perform bookkeeping operations.
  - Refresh
  - RowHammer protection
  - ...

Life of a DRAM Request

- Processor ticks, send read/write requests to the memory controller.
  - Controller::enqueue(req)

- Controller ticks:
  1. Serve completed read requests. Processor::receive(req)
  2. Ticks the refresh controller. Refresh::tick_ref()
  3. Query the scheduler to find a best request to issue its DRAM commands.
     - Controller::issue_cmd(cmd, addr_vec)
  4. Issue the command:
     I. Update the DRAM state machine, recursively through the entire hierarchy.
        - Controller::update_state(cmd, addr)
     II. Update the timing constraint for the DRAM commands, recursively.
         - Controller::update_timing(cmd, addr)
     III. Update the rowtable. RowTable::update(cmd, addr, clk)
template <typename T>
void DRAM<T>::update_state(T::Command cmd, const int* addr)
{
    int child_id = addr[int(level)+1];
    if (lambda[int(cmd)])
        lambda[int(cmd)](this, child_id); // update this level

    if (level == spec->scope[int(cmd)] || !children.size())
        return; // stop recursion: updated all levels

    // recursively update my child
    children[child_id]->update_state(cmd, addr);
}
template <typename T>
void DRAM<T>::update_state(typename T::Command cmd, const int* addr) 
{
    int child_id = addr[int(level)+1];
    if (lambda[int(cmd)]) lambda[int(cmd)](this, child_id); // update this level

    if (level == spec->scope[int(cmd)] || !children.size())
        return; // stop recursion: updated all levels

    // recursively update my child
    children[child_id]->update_state(cmd, addr);
}
template <typename T>
void DRAM<T>::update_state(typename T::Command cmd, const int* addr) {
    int child_id = addr[int(level)+1];
    if (lambda[int(cmd)])
        lambda[int(cmd)](this, child_id); // update this level

    if (level == spec->scope[int(cmd)] || !children.size())
        return; // stop recursion: updated all levels

    // recursively update my child
    children[child_id]->update_state(cmd, addr);
}

Terminate the recursion early if we have reached the scope of a command.
template <typename T>
void DRAM<T>::update_timing(typename T::Command cmd, const int* addr, long clk)
{
    // I am not a target node: I am merely one of its siblings
    if (id != addr[int(level)])
    {
        for (auto& t : timing[int(cmd)])
        {
            if (!t.sibling)
                continue; // not an applicable timing parameter

            long future = clk + t.val;
            next[int(t.cmd)] = max(next[int(t.cmd)], future); // update future
        }
    return; // stop recursion: only target nodes should be recursed
    }
// To be continued...
template <typename T>
void DRAM<T>::update_timing(Command cmd, const int* addr, long clk)
{
    // I am not a target node: I am merely one of its siblings
    if (id != addr[int(level)])
    {
        for (auto& t : timing[int(cmd)])
        {
            if (!t.sibling)
                continue; // not an applicable timing parameter

            long future = clk + t.val;
            next[int(t.cmd)] = max(next[int(t.cmd)], future); // update future
        }
    } // stop recursion: only target nodes should be recursed

    return; // To be continued...

    // Timing constraint entries from the DRAM spec
}
template <typename T>
void DRAM<T>::update_timing(typename T::Command cmd, const int* addr, long clk) {
    // I am not a target node: I am merely one of its siblings
    if (id != addr[int(level)]) {
        for (auto& t : timing[int(cmd)]) {
            if (!t.sibling)
                continue; // not an applicable timing parameter
            The earliest time that the command can be issued again in the future
            long future = clk + t.val;
            next[int(t.cmd)] = max(next[int(t.cmd)], future); // update future
        }
        return; // stop recursion: only target nodes should be recursed
    }
    // To be continued...
DRAM Timing Constraint

// I am a target node
if (prev[int(cmd)].size())
{
    // Update history
    prev[int(cmd)].pop_back();
    prev[int(cmd)].push_front(clk);
}
for (auto& t : timing[int(cmd)])
{
    if (t.sibling) continue; // not an applicable timing parameter
    long past = prev[int(cmd)][t.dist-1];
    if (past < 0) continue; // not enough history
    long future = past + t.val;
    next[int(t.cmd)] = max(next[int(t.cmd)], future);
    if (!children.size()) return;
    // recursively update all children
    for (auto child : children) child->update_timing(cmd, addr, clk);
}
DRAM Timing Constraint

```cpp
// I am a target node
if (prev[int(cmd)].size()) {
    // Update history
    prev[int(cmd)].pop_back();
    prev[int(cmd)].push_front(clk);
}
for (auto& t : timing[int(cmd)]) {
    if (t.sibling) continue; // not an applicable timing parameter
    long past = prev[int(cmd)][t.dist-1];
    if (past < 0) continue; // not enough history
    long future = past + t.val;
    next[int(t.cmd)] = max(next[int(t.cmd)], future);
    if (!children.size()) return;
    // recursively update all children
    for (auto child : children) child->update_timing(cmd, addr, clk);
}
```
Handles timing constraints involving multiple histories e.g., nFAW
Using Ramulator
Building Ramulator

- **Dependencies**
  - yaml-cpp: Ramulator uses yaml-cpp to parse configuration files. The repo includes yaml-cpp-0.7.0 as a git submodule in ext/yaml-cpp.
  - fmt: Ramulator uses fmt to provide string formatting with std::iostream. The repo includes fmt-8.1.1 as a git submodule in ext/fmt.
  - gcc: Ramulator requires a compiler with C++17 support (e.g., g++-8 and above).

- Clone the Ramulator repo with submodules with
  - git clone --recursive https://gitlab.ethz.ch/hluo/ramulator.git

- Configure and compile the Ramulator executable
  - mkdir build; cd build;
  - cmake ..; make -j;
  - cp ramulator ../../ramulator; cd ..
Getting Started

- Ramulator has all its configurations organized in a YAML file.
- To run a simulation, supply Ramulator with the option `--config/-c` followed by the path to the configuration file.
  - `./ramulator --config ./configs/DDR4.yaml`
- Ramulator also supports overriding simulation parameters through the command line interface without modifying the YAML configuration file.
  - `ramulator --config ./configs/DDR4.yaml \ `param` memory.spec.speed.nRCD=50`
  - Multiple parameters can be set by adding multiple `--p <param>=<value>` pairs.
To perform larger scale and more complicated parameter sweeps, it is recommended to use a Python script to drive the simulation.

- Parse the YAML configuration file (with the PyYAML library).
- Edit the parsed YAML (e.g., loop through a range of values for some parameters).

Example

```python
import yaml
import subprocess, sys

# Parse the YAML configuration file
ramulator_config=None with open("configs/DDR4.yaml") as f:
    ramulator_config = yaml.load(f, Loader=yaml.FullLoader)

# Sweep trace files
for trace in ["cputraces/401.bzip2", "cputraces/403.gcc", "cputraces/429.mcf", "cputraces/444.namd"]:  
    ramulator_config["processor"]["trace"][0] = trace

# Sweep nRCD latencies
for nRCD in [20, 30, 40, 50]:
    ramulator_config["memory"]("spec")["speed"]["nRCD"] = nRCD
    result = subprocess.run(["./ramulator", yaml.dump(ramulator_config)])
```
Configuration File Format

- Ramulator uses YAML as its configuration file format for both human- and machine-readable as well as extensibility.

- A YAML document is a collection of hierarchical key-value pairs ("nodes"). A key is a string (or hierarchically, a node), while its corresponding value can be one of the followings:
  - **Scalar**: Primitive types like integer, float, string.
  - **Mapping**: A collection (unordered set) of nodes without duplicate keys.
  - **Sequence**: A list (ordered series) of nodes that can have duplicate keys.

```
K0: V0
K1: V1
K2: V2
K:
  - S0
  - S1
  - S2

Mapping Sequence
```
Configuration File Format

- At a high level, a Ramulator configuration file must contain a memory node that includes all parameters about the memory subsystem.

- If there is also a processor node in the configuration file, Ramulator will run in the CPU-trace-driven mode.
  - Otherwise, Ramulator runs in the DRAM-trace-driven mode.

- Examples
Extending Ramulator
Key Components of Ramulator

- **Memory**: HW/SW interface between CPU/OS and DRAM controllers.
  - Translation: virtual-to-physical address translation.

- **Controller**: memory controller.
  - AddrMapper: physical-to-DRAM address mapping.
  - Scheduler: request scheduler.
  - Refresh: refresh control.
  - RowPolicy: row management policy (e.g., open-page, timeout).
  - RowTable: bookkeeping of the row status in each bank.

- **DRAM**: a DRAM node, implements the hierarchical state machine.
  - Traversal initiated in Controller::issue_cmd(cmd, addr_vec)

- DRAM standard definitions: Specifies the organization, timing constraints, state transitions, etc.
General Rules of Extending Ramulator

- Think about how to make your design and implementation modular.
  - In which part of the memory controller logic does the design belong to?
  - Is the design specific to a certain DRAM standard, or is it generally applicable to many DRAM standards?

- Implement your design by inheriting from the base class that it fits.
  - If the difference is small compared to the existing common implementation, inherit from it instead and only overwrite the involved class methods.
    - `Controller::issue_cmd(cmd, addr_vec)`
  - Or if it is specific to a certain DRAM standard, implement it as a specialization to the existing templated common implementation.

- Add your design to the factory methods of the corresponding base class.
  - For example, `SchedulerBase<T>* make_scheduler(const YAML::Node& config, Controller<T>* ctrl)`
Adding a New DRAM standard

- Starts with copying an existing DRAM standard (e.g., `DDR4.h` and `DDR4.cpp`).
- Add new levels of DRAM organization hierarchies (if any).
- Add new DRAM commands definitions and the corresponding state machine update/query functions.
- Add new timing parameter definitions and the corresponding timing constraints between DRAM commands at each level in the DRAM hierarchy.
- Implement any specialization to the memory controller components if necessary.
  - Or if the difference is small, try conditional compilation with `if constexpr` and type traits (C++17 features).
P&S Ramulator
Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator

Haocong Luo
Prof. Onur Mutlu
ETH Zürich
Spring 2022
18 March 2022