Computer Architecture

Lecture 22b: Ramulator 2.0

Haocong Luo
Prof. Onur Mutlu
ETH Zürich

Fall 2023

8 December 2023

Introduction

- **Haocong Luo**
  - PhD Student @ SAFARI research group
  - MS in Computer Science from ETH Zurich
- **Research Interests**
  - Memory system & DRAM
- **Contact**
  - haocong.luo@safari.ethz.ch

Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator

Haocong Luo
Prof. Onur Mutlu
ETH Zürich

Fall 2023

8 December 2023

Executive Summary

- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Architecture
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Executive Summary

- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Architecture
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Motivation

- We need cycle-accurate DRAM simulators to:
  - Model the operation of the memory controller and DRAM in detail
  - Evaluate the performance, energy efficiency, security, etc. of the memory system
- Growing research and design efforts to improving the performance, security, and reliability of DRAM-based memory systems
  - We need a tool to enable rapid and agile implementation and evaluation of new designs

Motivation (II)

- Problems of existing DRAM simulators: Not modeling the memory system in a modular and extensible way
  - Issue 1: Not separating the memory controller logic from the behavioral and timing model of the DRAM device
  - Issue 2: Lacking a concise and intuitive way to implement DRAM specifications

Executive Summary

- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Architecture
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Goal

- Provide an easy-to-use, modular, and extensible software infrastructure for rapid and agile implementation and evaluation of DRAM-related research and design ideas
  - Should have a modular and extensible software architecture
    - Changing the functionality of one component should not involve code changes of other unrelated components
  - Should facilitate easy modification of DRAM specifications (e.g., DRAM organization, commands, timing constraints)
    - Requires an intuitive and concise way to define and implement these

Executive Summary

- Motivation
- Goal
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques

Ramulator 2.0

- Fundamentally modular and extensible software framework
  - Every component in the memory system is modeled by an abstract interface and concrete implementations
- Concise and human-readable DRAM specifications
  - Intuitive definition with string literals, easy to understand (modify)
- DRAM models for new (and old) standards
  - DDR3, DDR4, DDR5, LPDDR5, GDDR6, HBM1, HBM2, HBM3, …
- A variety of RowHammer mitigation techniques
  - PARA, TWiCe, Graphene, Hydra, RRS, …
- Fast simulation speed (using some C++20 features)
- Can either work as a standalone simulator or be used as a memory system library by a system simulator (e.g., gem5)
- Public open-source version: https://github.com/CMU-SAFARI/ramulator2

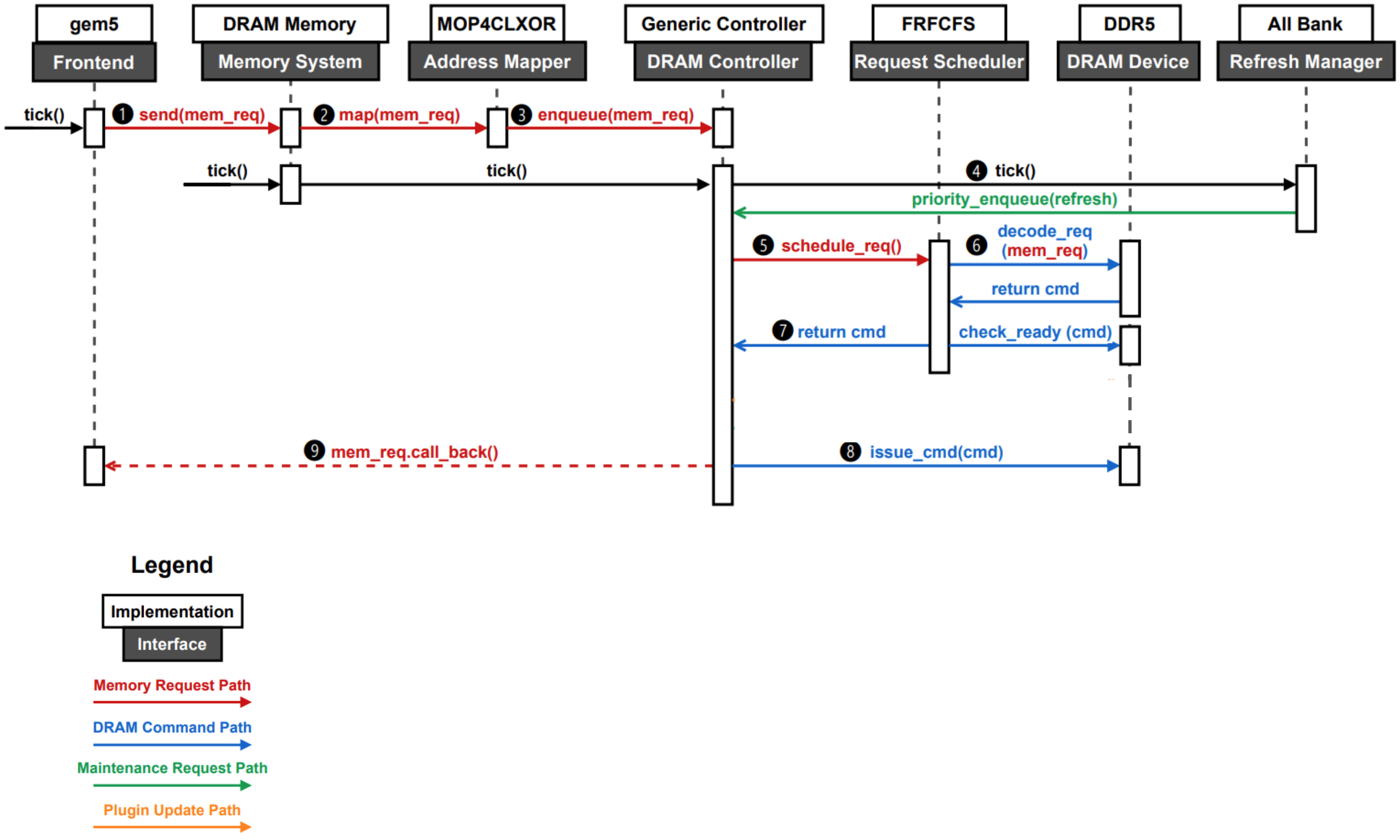
Executive Summary

- Motivation
- Goal
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques
- Conclusion

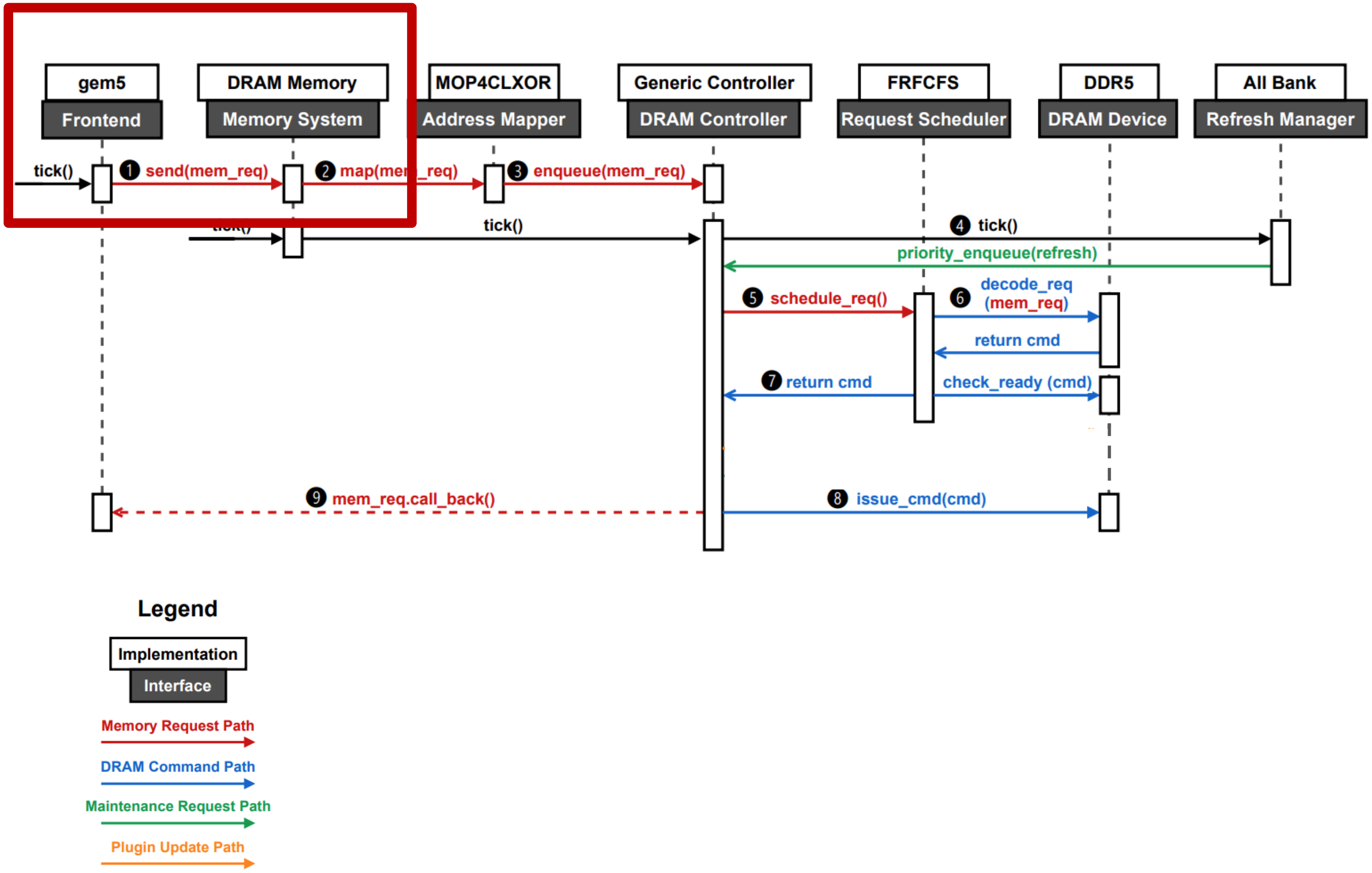
Modular and Extensible Software Framework

- Recall: We want the simulator to be modular and extensible
  - Changing the functionality of one component should not involve code changes of other unrelated components
- Solution: Decouple how a component interacts with other components in the system from the functionality of its own
  - Interface: An abstract C++ class that models the common high-level behavior of a component as seen by other components
  - Implementation: A concrete C++ class that inherits an interface and models the actual functionality of a component
- Example:

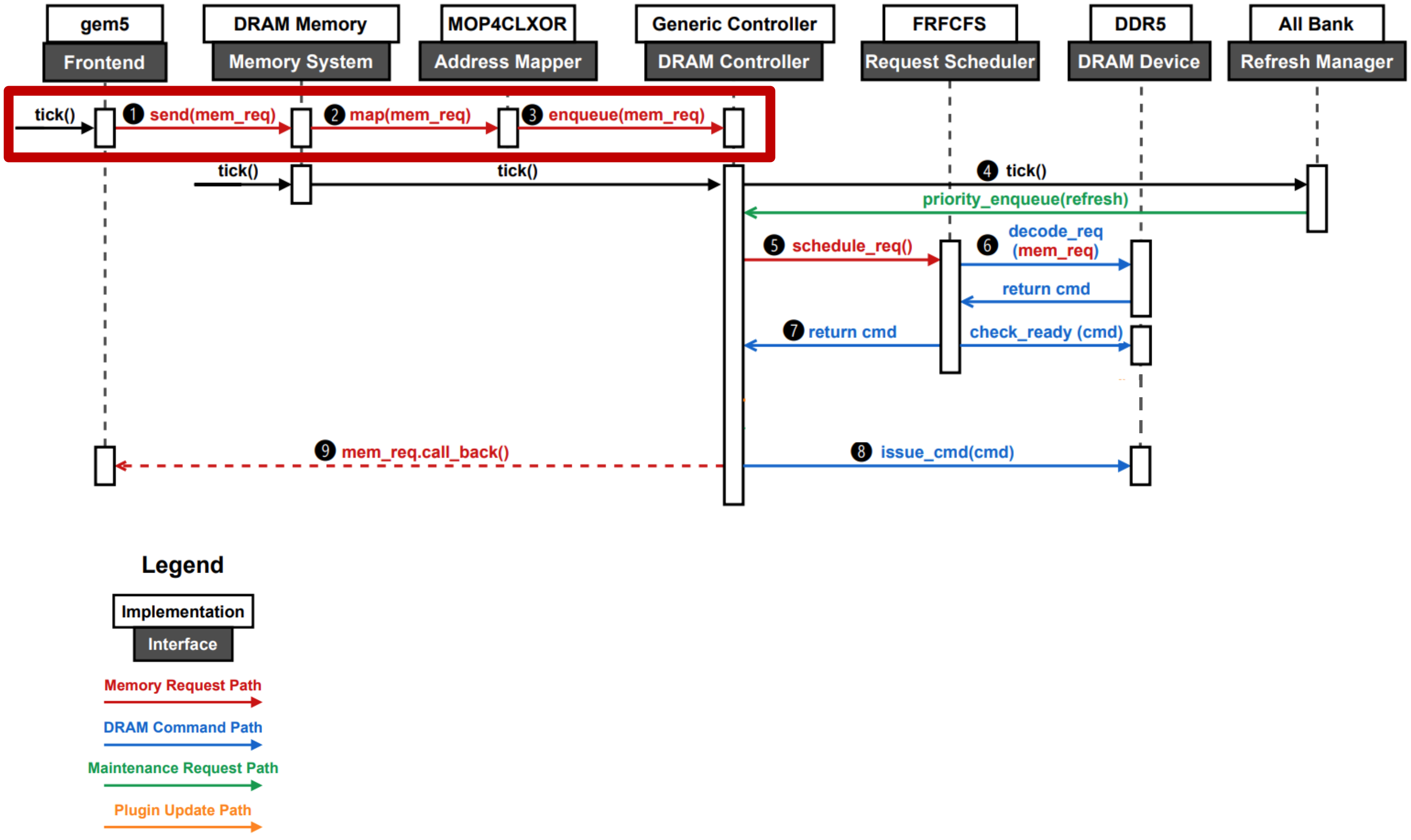
An Example Memory System Configuration



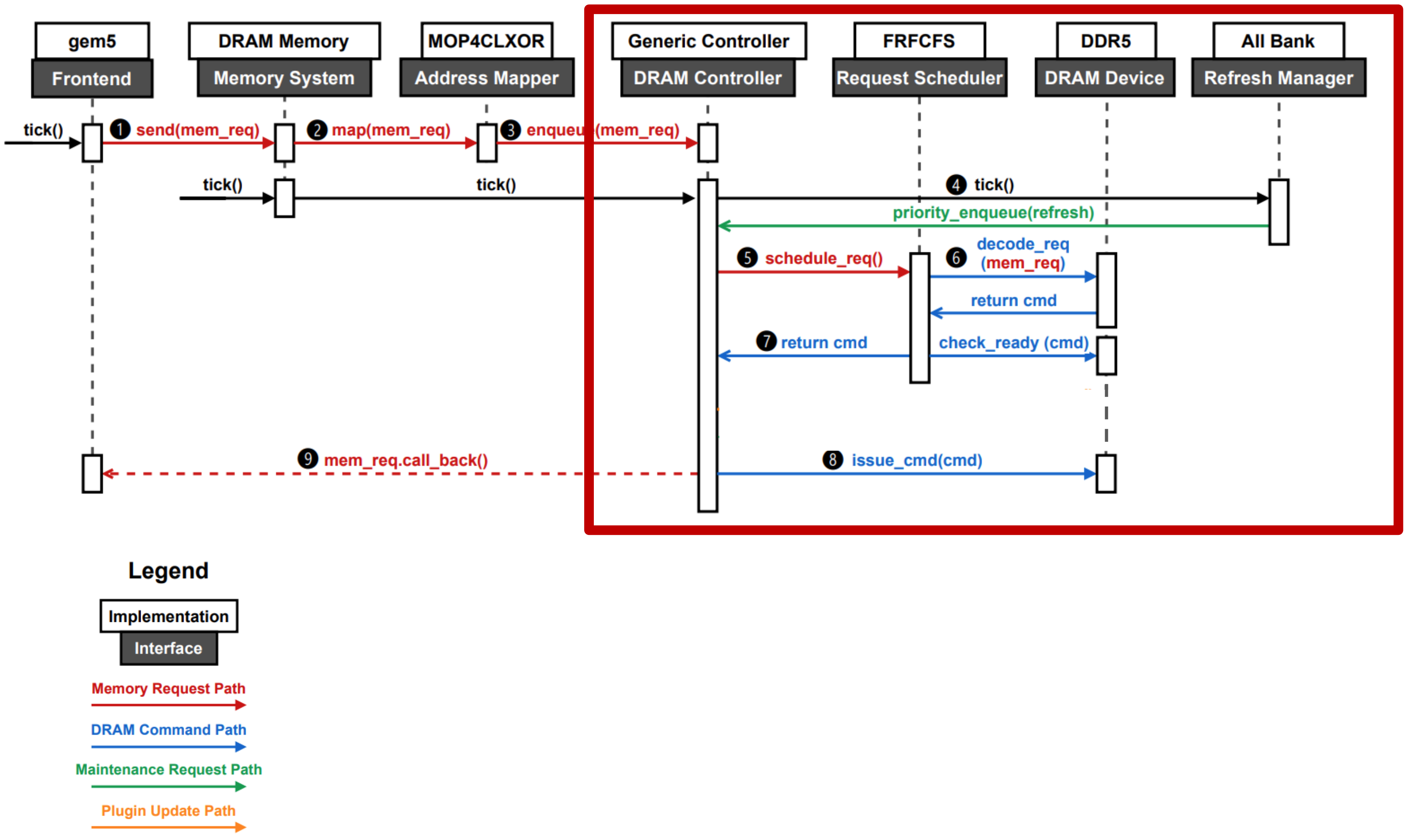
An Example Memory System Configuration



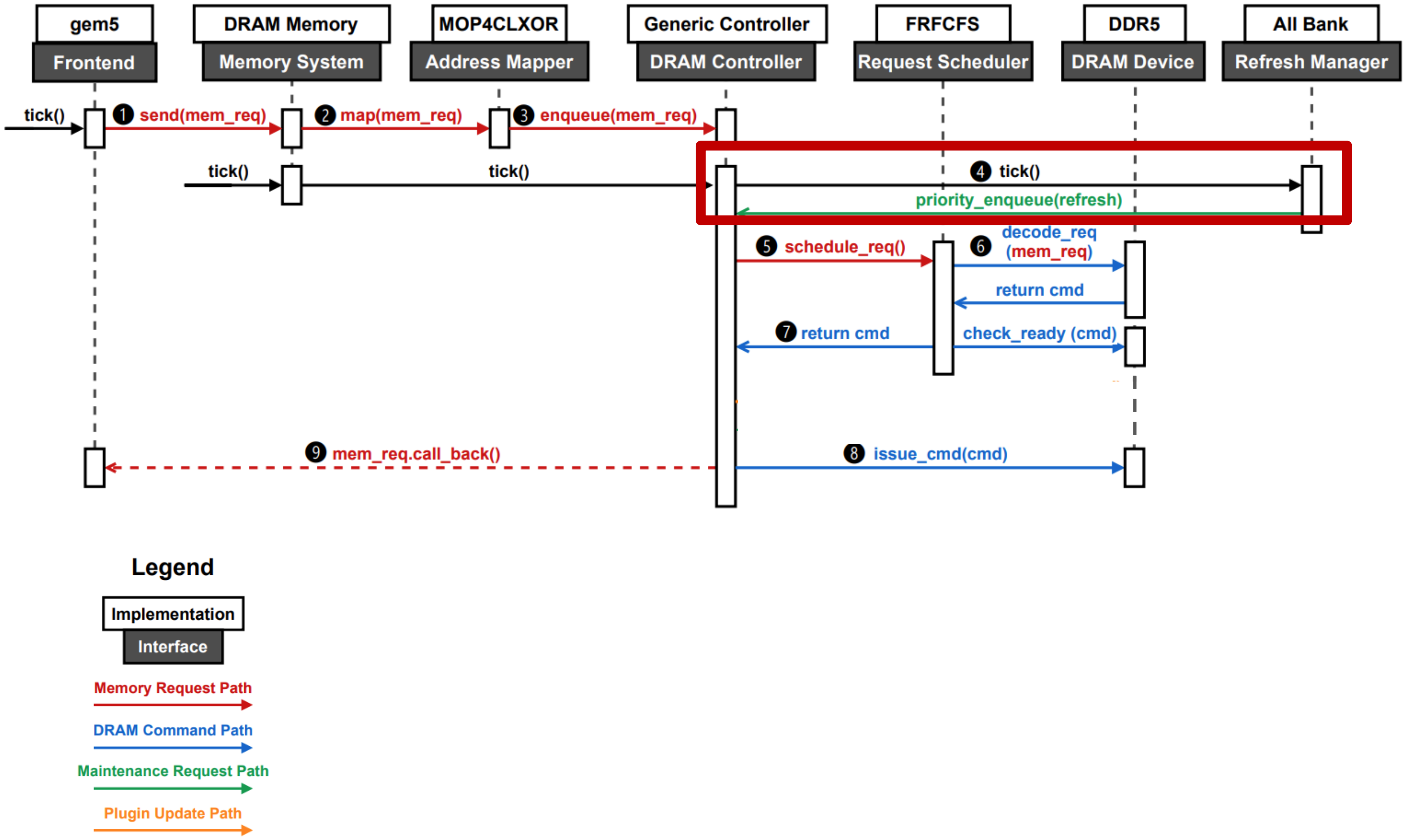
An Example Memory System Configuration



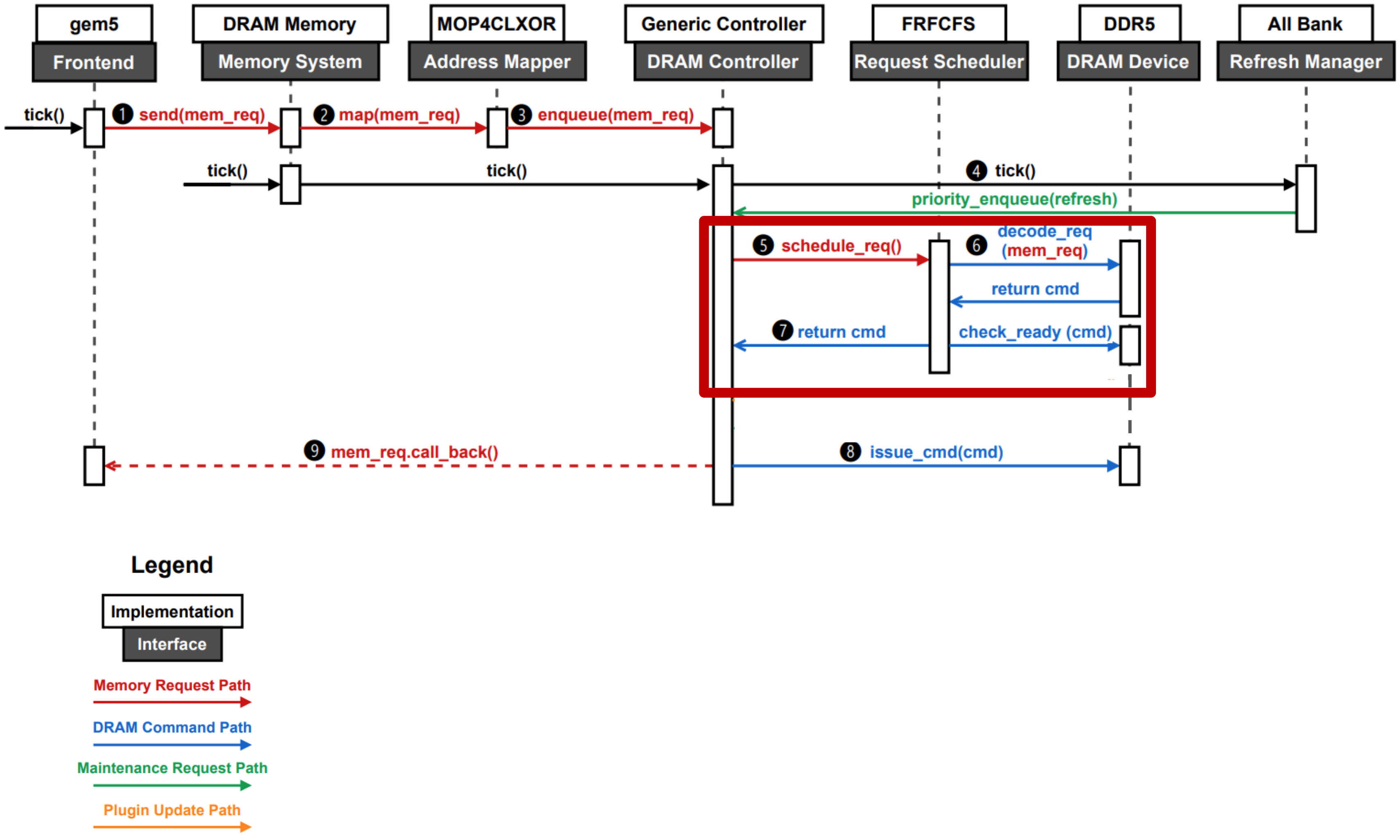
An Example Memory System Configuration



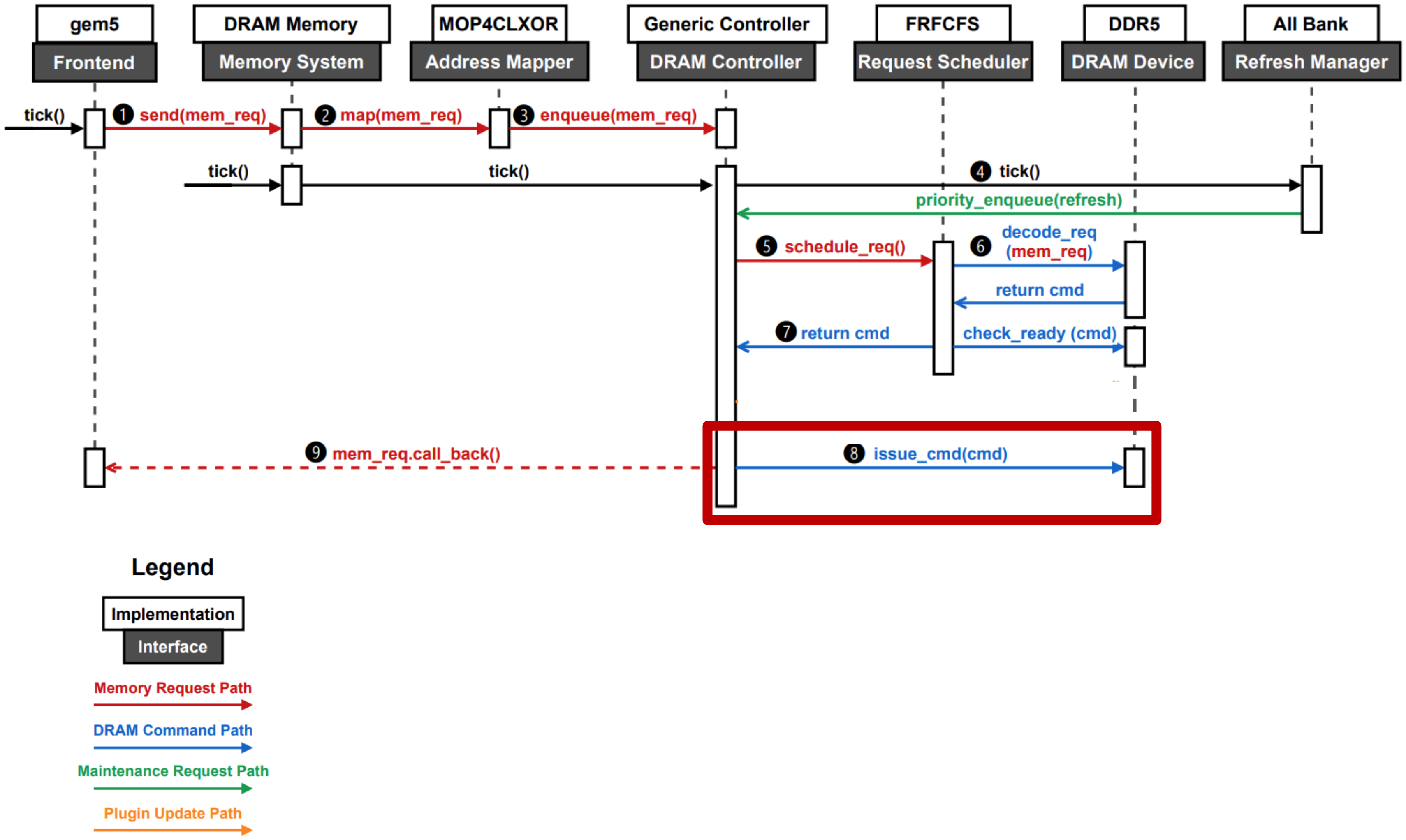
An Example Memory System Configuration



An Example Memory System Configuration



An Example Memory System Configuration



Modular and Extensible Software Framework

- Recall: We want the simulator to be modular and extensible
  - Changing the functionality of one component should not involve code changes of other unrelated components
- Solution: Decouple how a component interacts with other components in the system from the functionality of its own
  - Interface: An abstract C++ class that models the common high-level behavior of a component as seen by other components
  - Implementation: A concrete C++ class that inherits an interface and models the actual functionality of a component
- Question: How do you efficiently manage all the interfaces and implementations?
  - Ramulator 2.0 implements a self-registering factory to keep track all the interfaces and their implementations
  - Automatically instantiates implementations by their names from the configuration file

Executive Summary

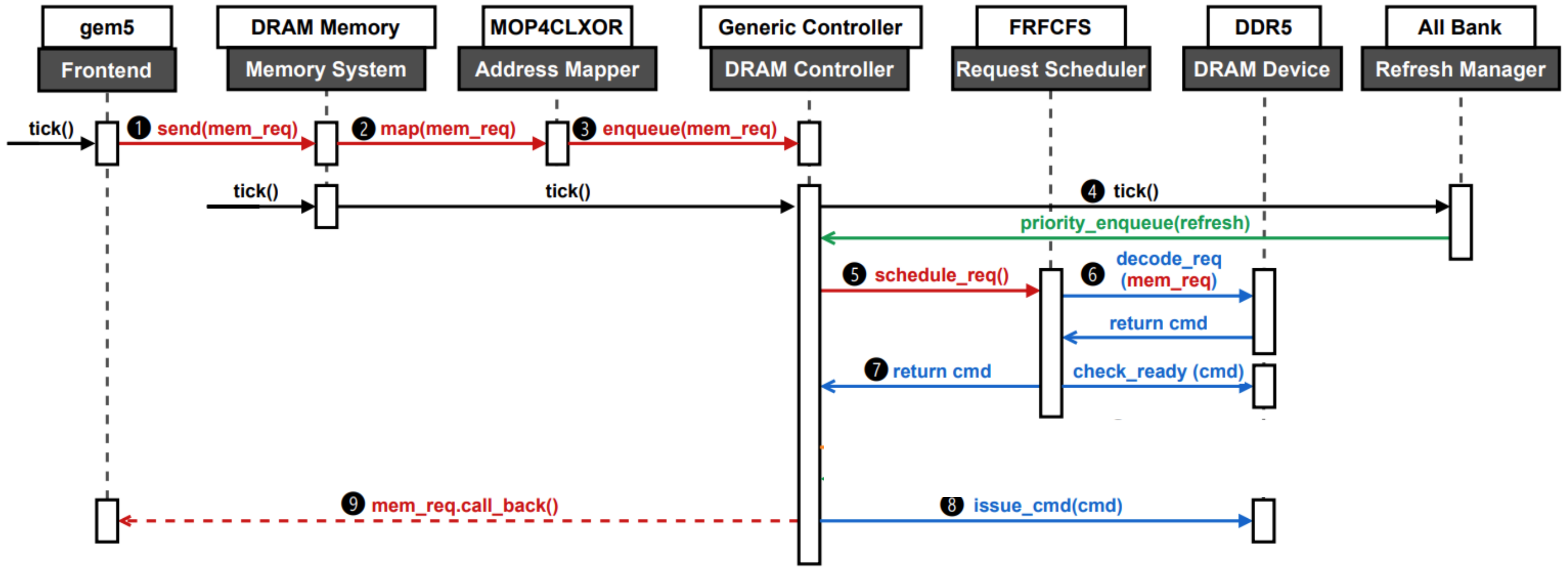
- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Memory Controller Plugins

- The memory controller has a lot of responsibilities:
  - Decode memory requests into DRAM commands based on the current state and timing constraints of the DRAM device
  - Schedule memory requests based on the scheduling policy
  - Perform maintenance and bookkeeping duties (e.g., periodic refresh)
- It is also where a lot of extra functionalities could be implemented:
  - Memory controller based RowHammer mitigations that tracks and records row activations
  - Utility functions that inspects the issued DRAM commands to:
    - Collect performance statistics and give feedback to the system
    - Provide insights for post-simulation analyses
- Does it make sense to have a different memory controller implement for every single such extra functionalities?

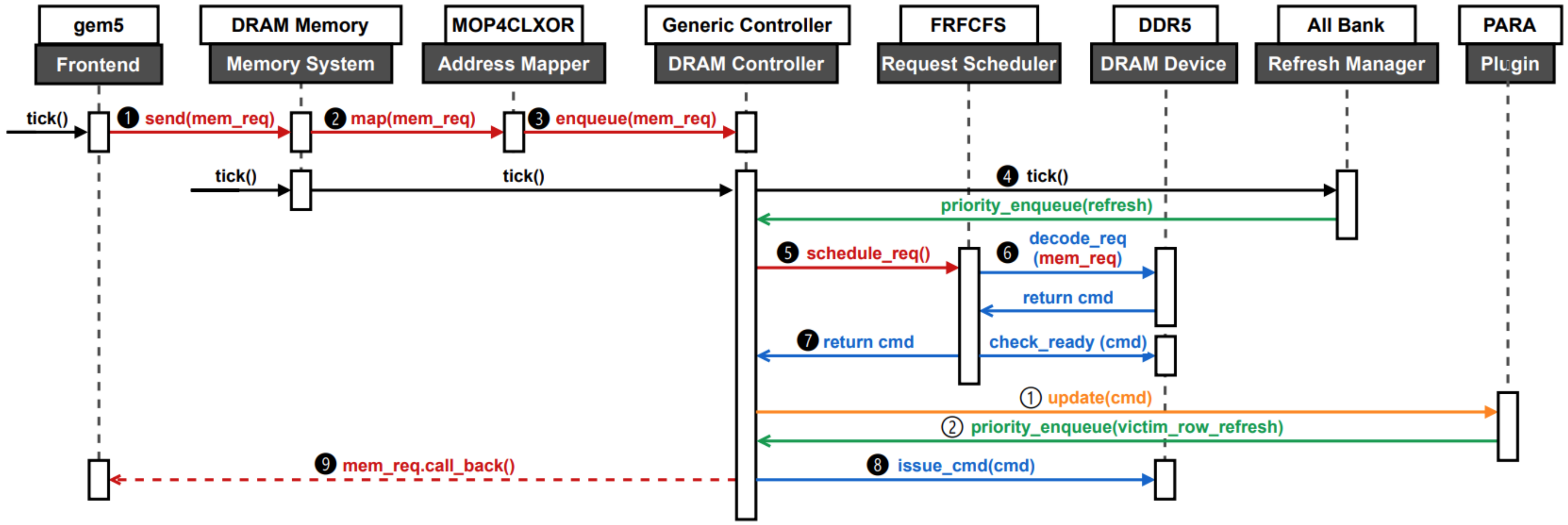
Memory Controller Plugins

- Does it make sense to have a different memory controller implement for every single such extra functionalities? No.
- Design a memory controller plugin interface for a generic memory controller implementation:
  - Change the plugin implementation to change its function without modifying the memory controller implementation



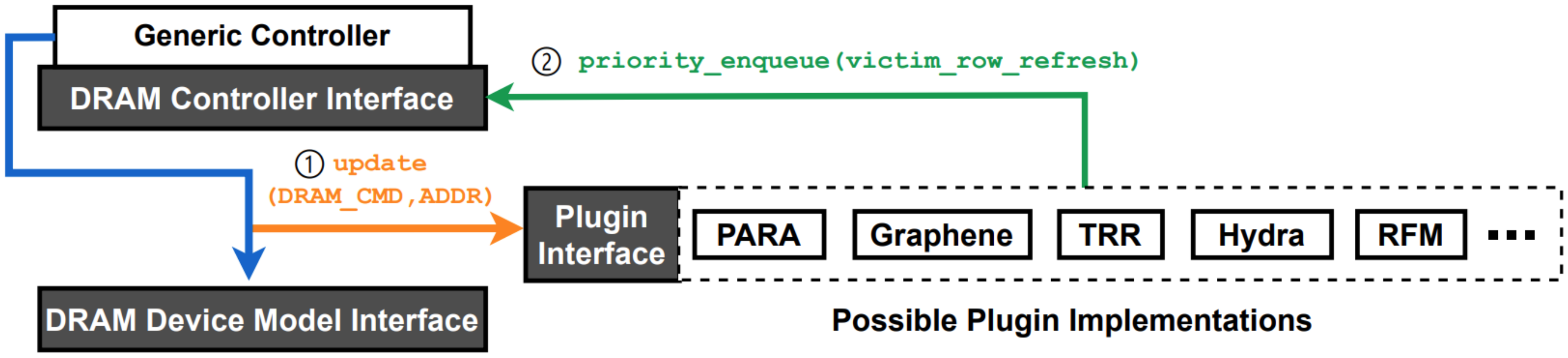
Memory Controller Plugins

- Does it make sense to have a different memory controller implement for every single such extra functionalities? No.
- Design a memory controller plugin interface for a generic memory controller implementation:
  - Change the plugin implementation to change its function without modifying the memory controller implementation



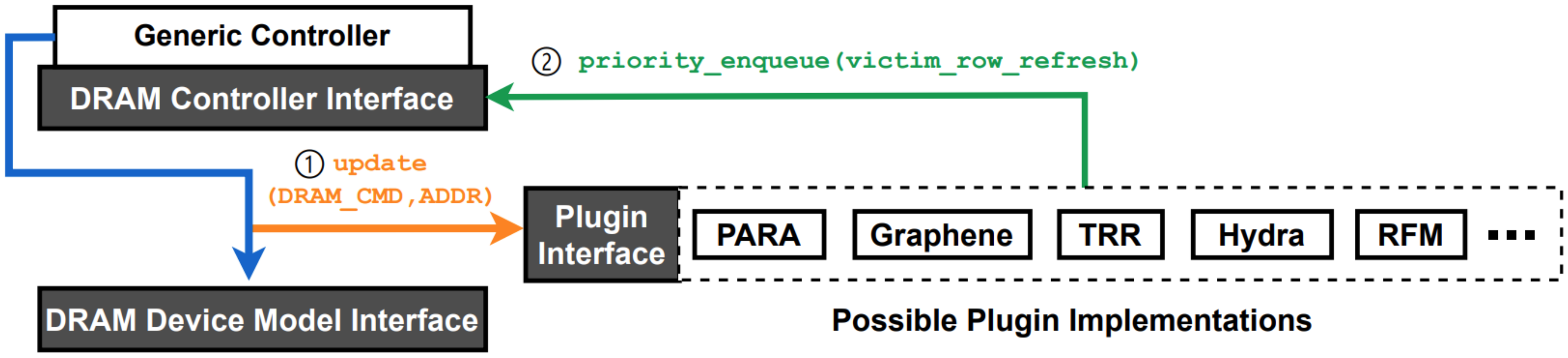
Memory Controller Plugins

- Example plugin implementations:
  - Memory controller based RowHammer mitigation techniques


Generic Controller
DRAM Controller Interface
② priority_enqueue(victim_row_refresh)
① update (DRAM_CMD,ADDR)
Plugin Interface
PARA
Graphene
TRR
Hydra
RFM
...
DRAM Device Model Interface
Possible Plugin Implementations

Memory Controller Plugins

- Example plugin implementations:
  - Memory controller based RowHammer mitigation techniques


Generic Controller
DRAM Controller Interface
② priority_enqueue(victim_row_refresh)
① update (DRAM_CMD,ADDR)
Plugin Interface
PARA
Graphene
TRR
Hydra
RFM
...
DRAM Device Model Interface
Possible Plugin Implementations

  - Useful utilities for debugging and subsequent analysis:
    - Dump DRAM command trace
    - Count the total number of commands
    - ...

Executive Summary

- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

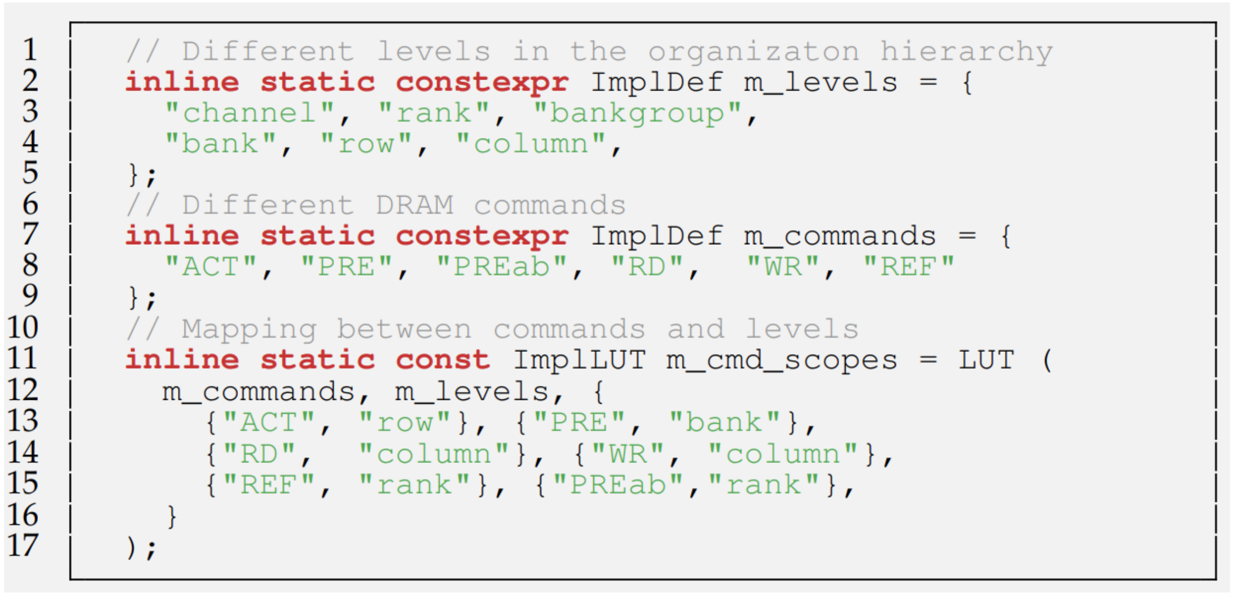
Concise and Intuitive DRAM Specifications

- The DRAM behavioral and timing models in Ramulator 2.0 are based on finite-state machines, implemented by many lookup tables (LUTs)
- Defining DRAM specifications (e.g., the organization of the DRAM device hierarchy, DRAM commands, mapping between DRAM commands and organization levels, timing constraints) are essentially filling the LUTs
- Goal:
  - Enable a concise and human-readable way to fill the LUTs (without runtime overhead)
  - Maintain modularity and extensibility

Concise and Intuitive DRAM Specifications

- Three key implementations:

  1) String-literal based definition of DRAM specifications



```
// Different levels in the organizaton hierarchy
inline static constexpr ImplDef m_levels = {
  "channel", "rank", "bankgroup",
  "bank", "row", "column",
};
// Different DRAM commands
inline static constexpr ImplDef m_commands = {
  "ACT", "PRE", "PREab", "RD",  "WR", "REF"
};
// Mapping between commands and levels
inline static const ImplLUT m_cmd_scopes = LUT (
  m_commands, m_levels, {
    {"ACT", "row"}, {"PRE", "bank"},
    {"RD",  "column"}, {"WR", "column"},
    {"REF", "rank"}, {"PREab","rank"},
  }
);
```

  - String-literals are const-evaled within the DRAM model to avoid runtime cost
  - Can also be dynamically queried by other components through the DRAM interface to achieve modularity

Concise and Intuitive DRAM Specifications

- Three key implementations:

  2) Permutation-based definition of timing constraints

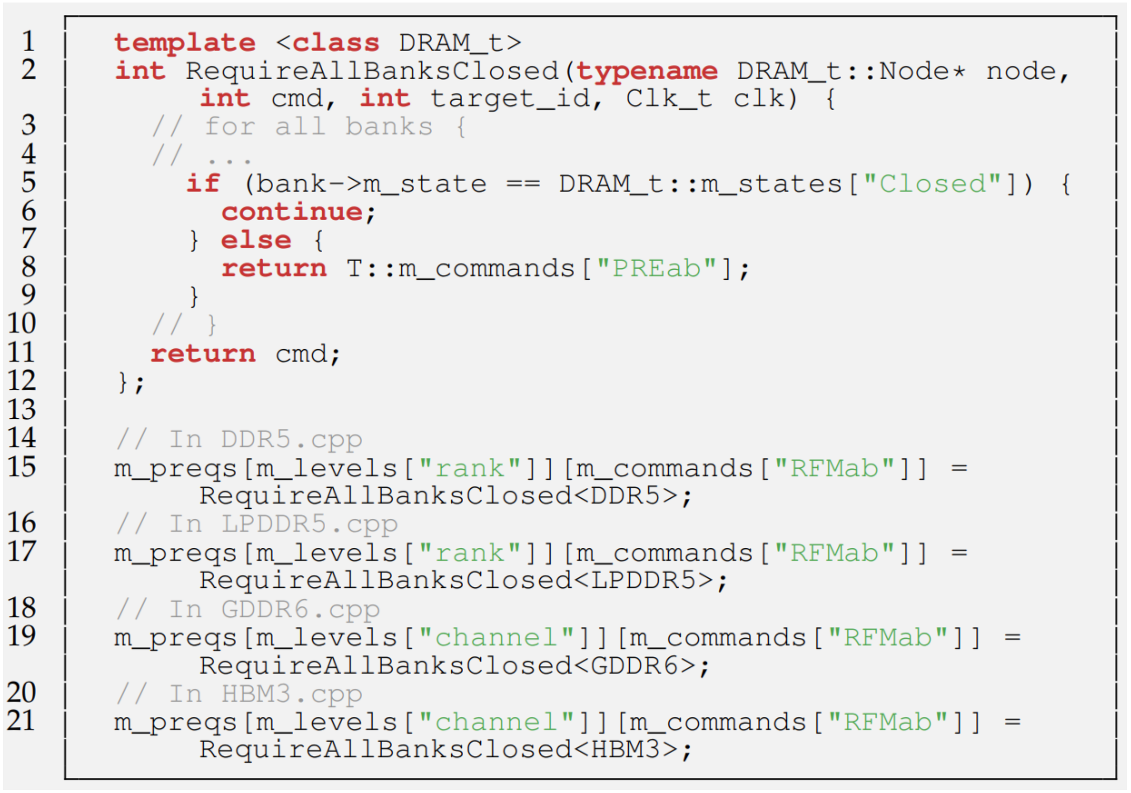
```
  {.level = "bank",
   .preceding = {"ACT"}, .following = {"RD", "WR"},
   .latency = V("nRCD")},
```

  - Defines that at the “bank” level, the minimum time interval from any of the “preceding” DRAM commands to any of the “following” commands is “nRCD” cycles
  - Automatically adds the corresponding timing constraint entries to the LUTs

Concise and Intuitive DRAM Specifications

- Three key implementations:

  3) Reusable library of DRAM command implementations



```
template <class DRAM_t>
int RequireAllBanksClosed(typename DRAM_t::Node* node,
      int cmd, int target_id, Clk_t clk) {
  // for all banks {
  // ...
    if (bank->m_state == DRAM_t::m_states["Closed"]) {
      continue;
    } else {
      return T::m_commands["PREab"];
    }
  // }
  return cmd;
};

// In DDR5.cpp
m_preqs[m_levels["rank"]][m_commands["RFMab"]] =
      RequireAllBanksClosed<DDR5>;
// In LPDDR5.cpp
m_preqs[m_levels["rank"]][m_commands["RFMab"]] =
      RequireAllBanksClosed<LPDDR5>;
// In GDDR6.cpp
m_preqs[m_levels["channel"]][m_commands["RFMab"]] =
      RequireAllBanksClosed<GDDR6>;
// In HBM3.cpp
m_preqs[m_levels["channel"]][m_commands["RFMab"]] =
      RequireAllBanksClosed<HBM3>;
```

  - A single templated implementation of the RFM command reused by many different DRAM standards

Executive Summary

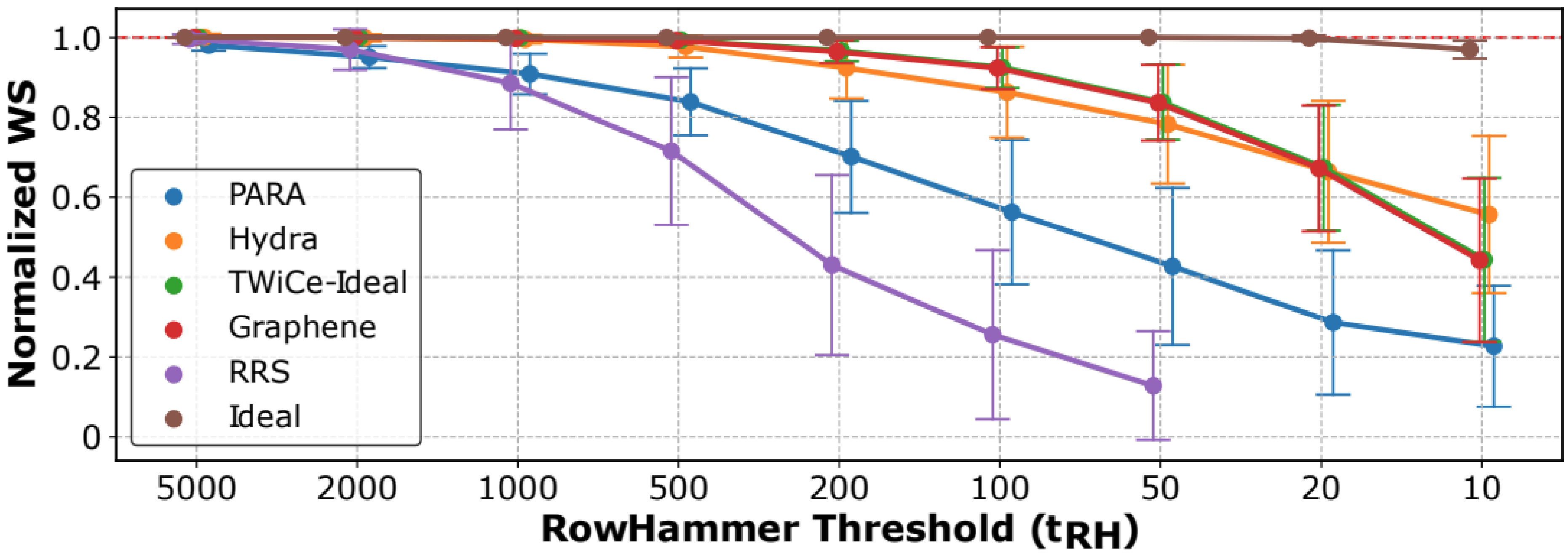
- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Cross-Sectional Study of Different RowHammer Mitigation Techniques

- Implement six different RowHammer mitigation techniques all as memory controller plugins:
  - PARA, an idealized version of TWiCe, Graphene, PARA, Hydra, Randomized RowSwap (RRS), and an idealized mitigation with infinite row activation count tracking capability (Ideal)
- Configure them for varying RowHammer threshold (i.e., the minimum number of DRAM row activations to cause at least one bitflip, tRH) from 5000 down to 10
- Evaluate the their performance overhead, measured by weighted speedup (WS), with 25 four-core multi-programmed benign workloads from SPEC06 and SPEC17

Cross-Sectional Study of Different RowHammer Mitigation Techniques

- Key Results:



Executive Summary

- **Motivation**
- **Goal**
- **Key Design Features of Ramulator 2.0**
  - Modular and Extensible Software Framework
  - Memory Controller Plugins
  - Concise and Intuitive DRAM Specifications
- **Example Use Case: Cross-Sectional Study of Different RowHammer Mitigation Techniques**
- **Conclusion**

Conclusion

- Ramulator 2.0 is a modern, modular, and extensible DRAM simulator, the successor to Ramulator 1.0
  - We introduce and demonstrate its key design features that enables high modularity and extensibility
- We hope that Ramulator 2.0’s modular and extensible software architecture and concise and intuitive modeling of DRAM facilitates more agile memory systems research
- More in our paper: https://arxiv.org/abs/2308.11030
- Public open-source version: https://github.com/CMU-SAFARI/ramulator2

Advertisement - Semester Projects

- Available Semester Projects with Ramulator 2.0
  - Implementing and verifying new DRAM standards
  - Implementing and verifying new memory technologies (e.g., STT-MRAM, FeRAM)
  - Incorporate a proper test framework for key components of Ramulator 2.0 (e.g., the DRAM models, the request scheduler) to enable regression tests
  - Implementing and evaluating hybrid memory systems
  - Implementing and evaluating memory request scheduling algorithms
  - Propose your new ideas in improving the performance, reliability, and energy efficiency of memory systems
  - …
- Contact
  - haocong.luo@safari.ethz.ch