# CS 758: Advanced Topics in Computer Architecture

Lecture #8: GPU Warp Scheduling Research + DRAM Basics

Professor Matthew D. Sinclair

Some of these slides were developed by Tim Rogers at the Purdue University, Tor Aamodt at the University of British Columbia, Wen-mei Hwu & David Kirk at the University of Illinois at Urbana-Champaign, Sudhakar Yalamanchili Georgia Tech, and Prof. Onur Mutlu at Carnegie Mellon University.

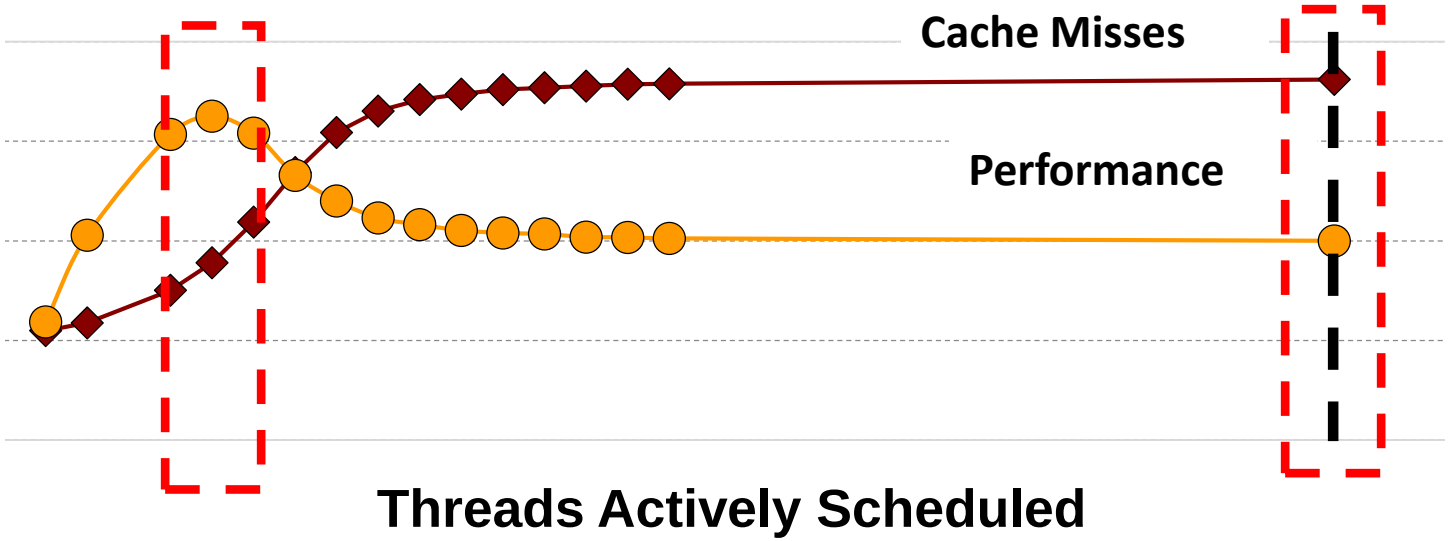
Slides enhanced by Matt Sinclair

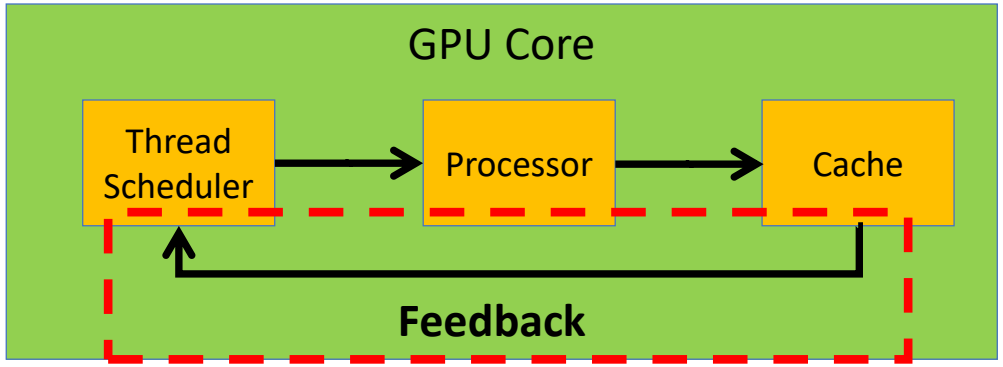
# Studying Warp Scheduling on GPUs

- Numerous works on manipulating different schedulers
- Most looking at the SM-side issue-level warp scheduler
- Some look at TB scheduling at the TB-core level
- Fetch scheduler and operand collector schedule less studied
  - Fetch largely follows issue.
  - Not clear what the opportunity in the operand collector is.
    - Even an opportunity study here would be helpful.

# Use Memory System Feedback

[MICRO 2012]



# Programmability case study [MICRO 2013]

## Sparse Vector-Matrix Multiply

GPU-Optimized Version
SHOC Benchmark Suite
(Oakridge National Labs)

Simple Version

**Example 2** GPU-Optimized SPMV-Vector Kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;
    int myRow = (blockIdx.x * warpsPerBlock)

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

                                    miters[myRow];
                                    ters[myRow+1];
    for (int j = warpStart + id;
         j < warpEnd; j += warpSize)
    {
        int col = cols[j];
        mySum += val[j] * vecTexReader(col);
    }
    partialSums[t] = mySum;

    // Reduce partial sums
    if (id < 16)
        partialSums[t] += partialSums[t+16];
    if (id <  8)
        partialSums[t] += partialSums[t+ 8];
    if (id <  4)
        partialSums[t] += partialSums[t+ 4];
    if (id <  2)
                                    tialSums[t+ 2];
                                    tialSums[t+ 1];

                                    Sums[t];
```

**Example 1** Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * bl
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Load
            int col = cols[j];
            t += val[j] * vecTe

        }
        out[myRow] = t;
    }
}
```

Explicit Scratchpad Use

Dependent on Warp Size

Added Complication

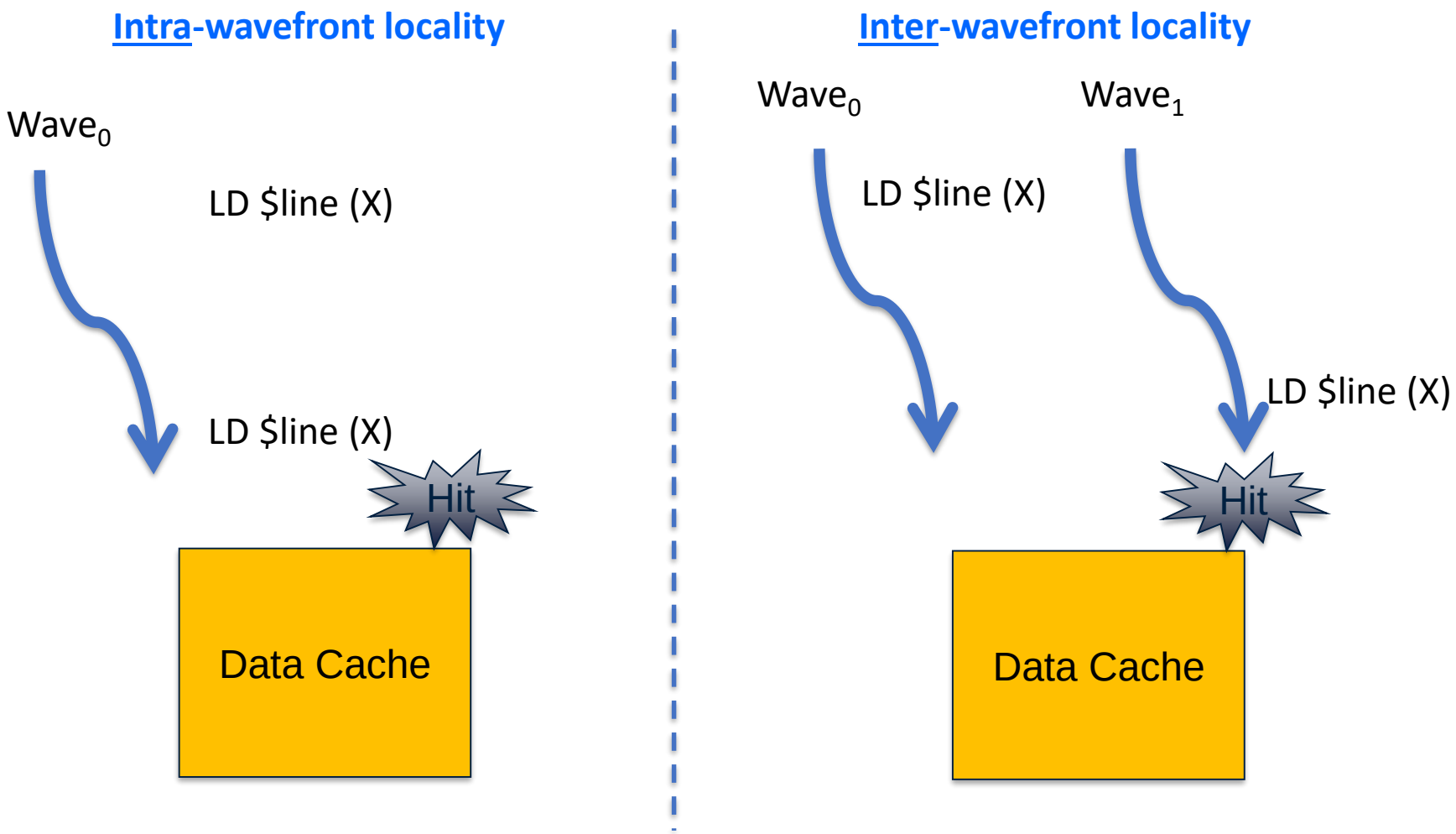
Parallel Reduction

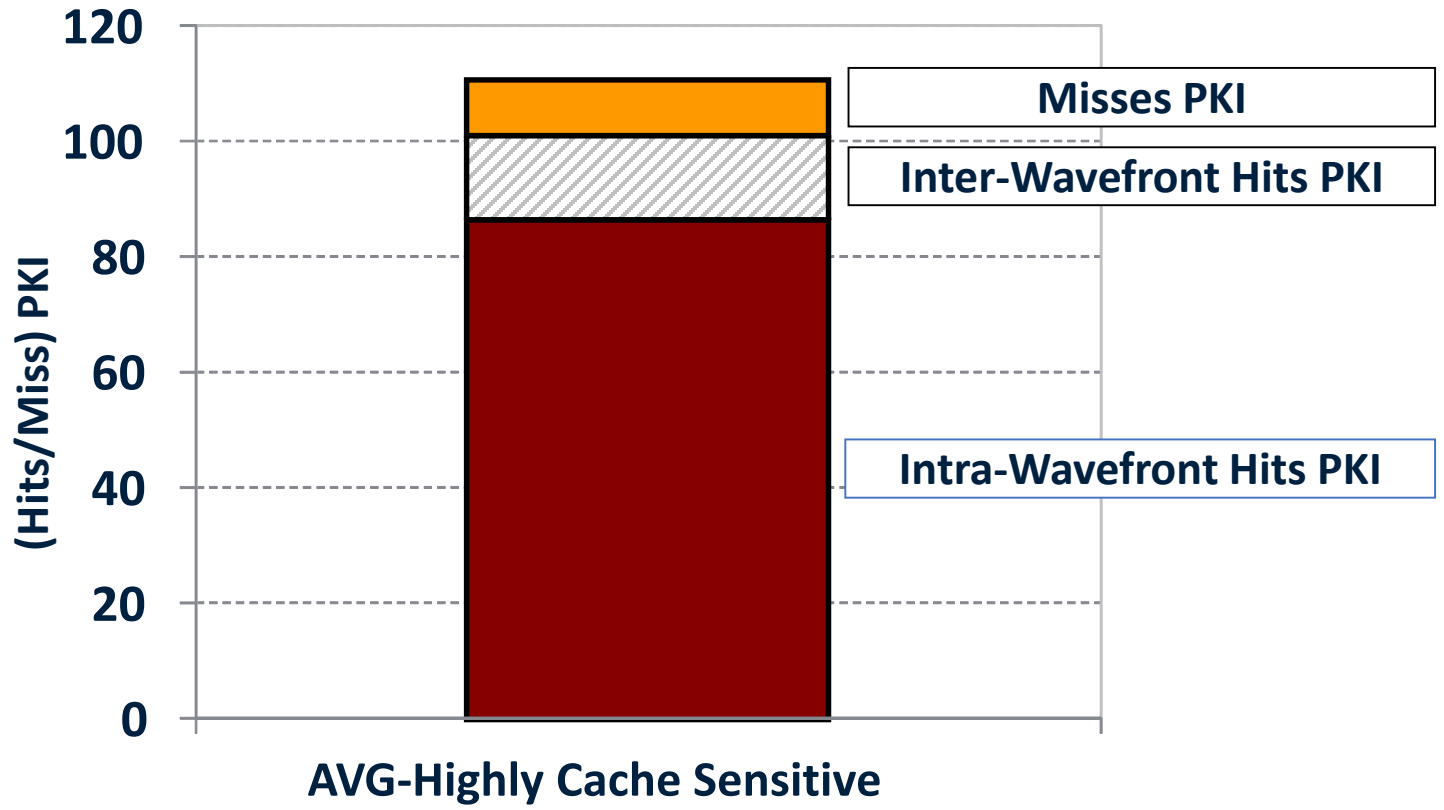
Divergence

Each thread has locality

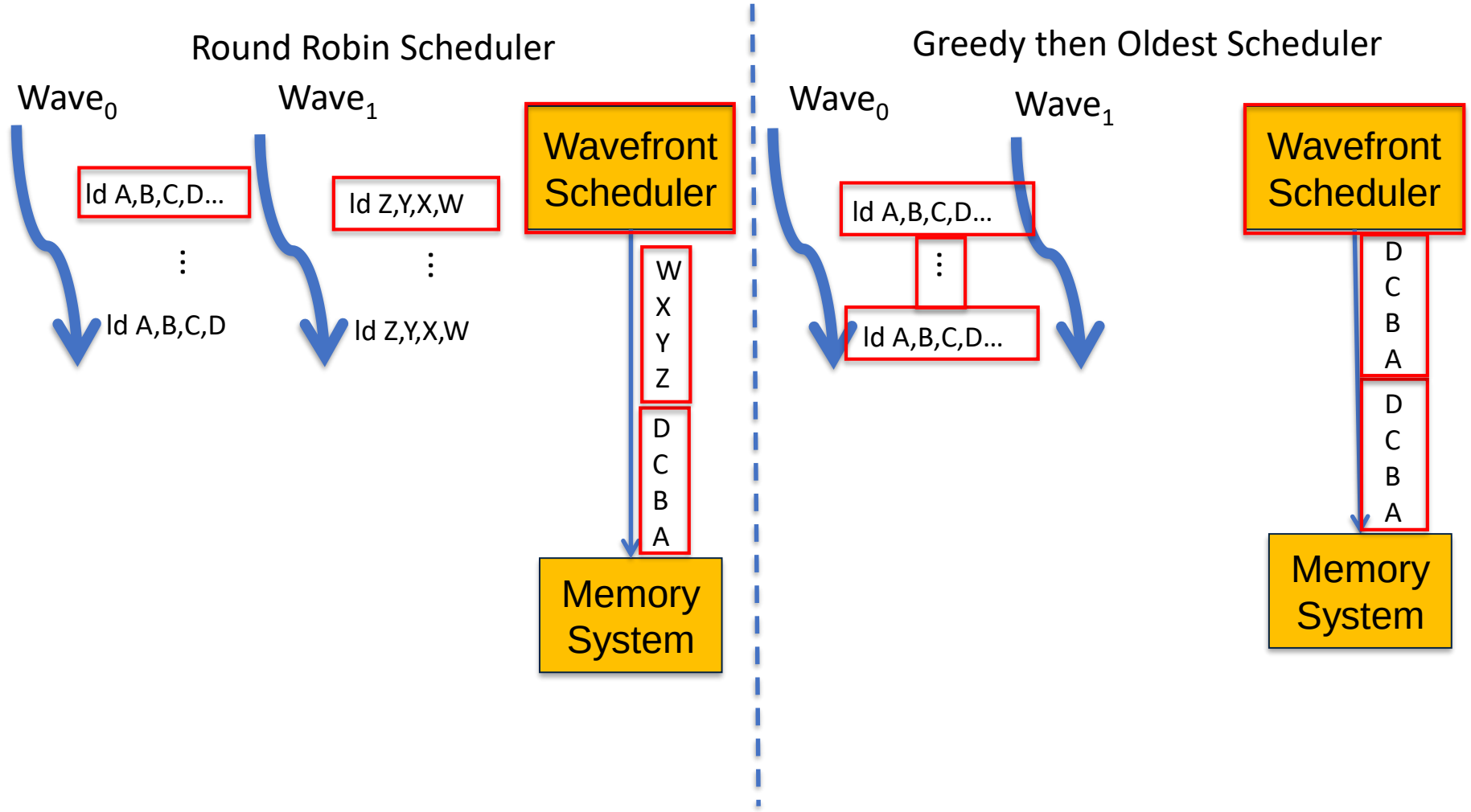

Using DAWS scheduling within 4% of optimized with no programmer input

# Sources of Locality

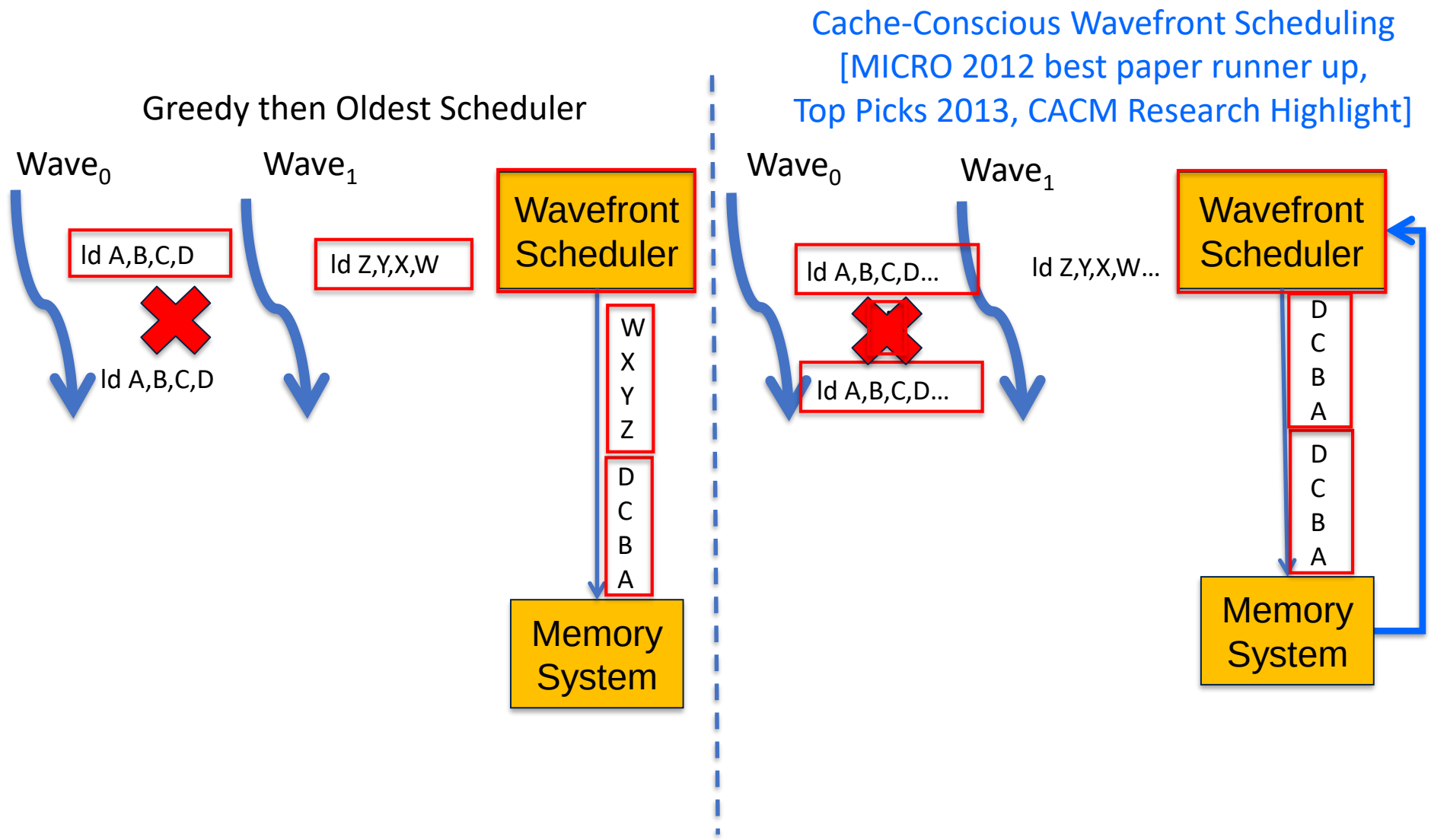


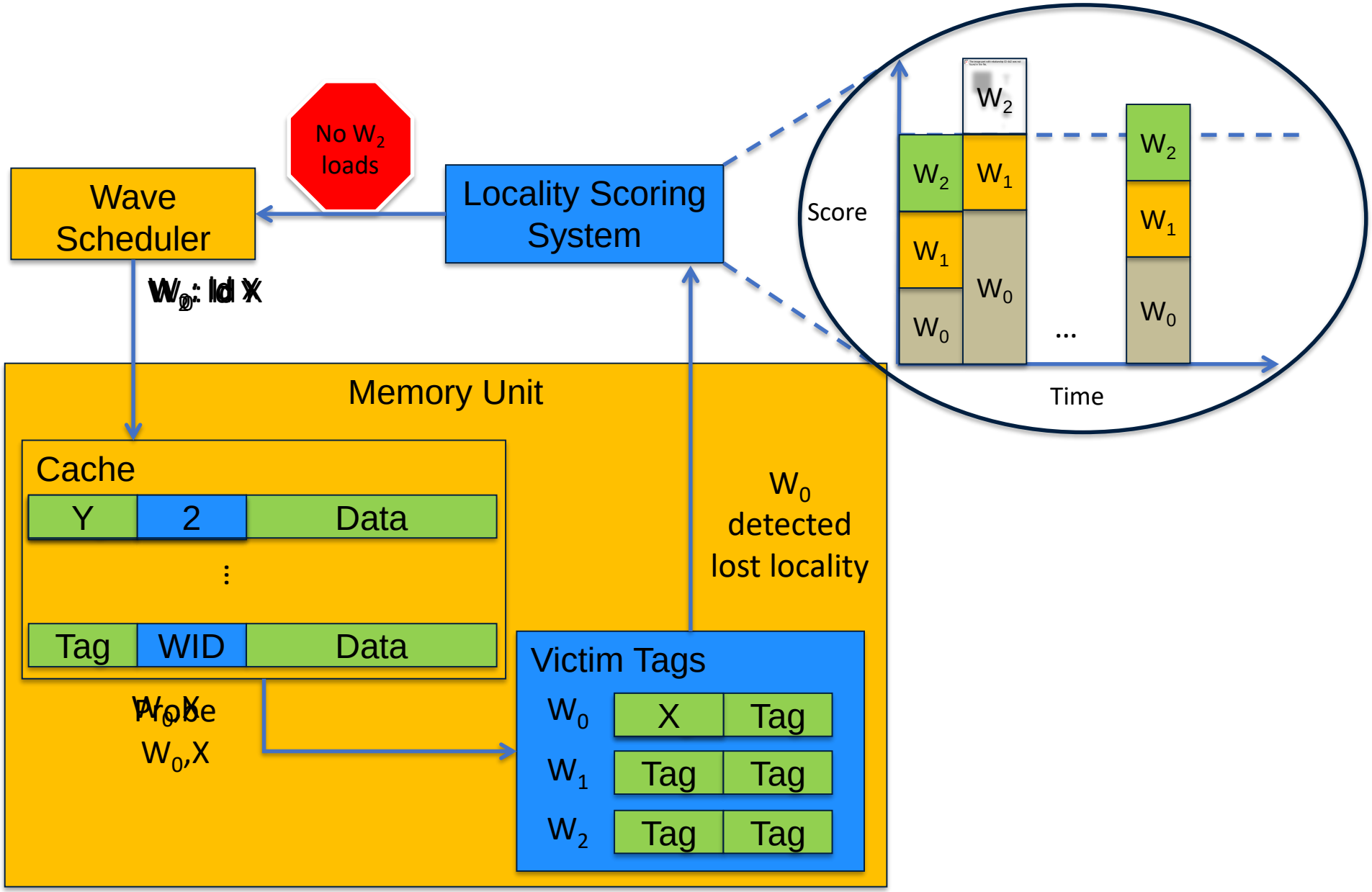

120
100
80
60
40
20
0
(Hits/Miss) PKI
Misses PKI
Inter-Wavefront Hits PKI
Intra-Wavefront Hits PKI
AVG-Highly Cache Sensitive

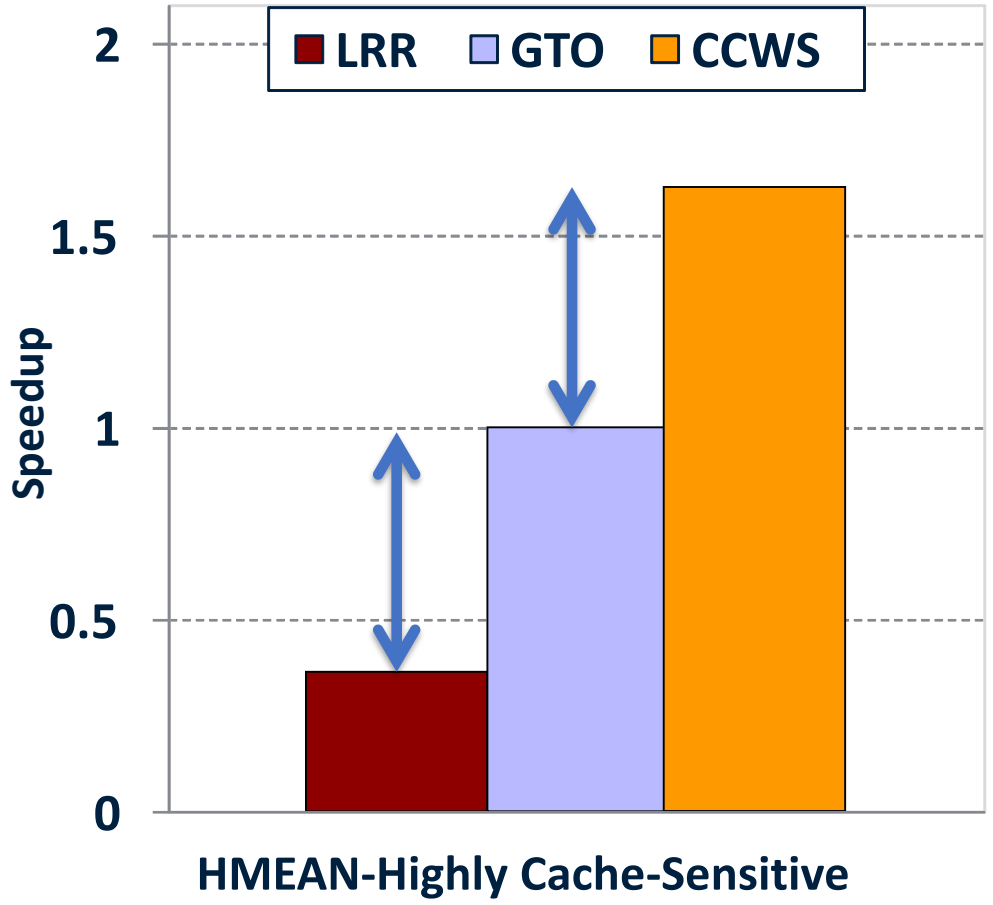
# Scheduler affects access pattern



# Use scheduler to *shape* access pattern




No $W_2$ loads
Wave Scheduler
Locality Scoring System
Score
Time
$W_2$
$W_1$
$W_0$
...
Memory Unit
Cache
Y
2
Data
Tag
WID
Data
$W_0$ detected lost locality
Victim Tags
$W_0$
X
Tag
$W_1$
Tag
Tag
$W_2$
Tag
Tag


LRR
GTO
CCWS
2
1.5
1
0.5
0
Speedup
HMEAN-Highly Cache-Sensitive

# Static Wavefront Limiting
[Rogers et al., MICRO 2012]

- Profiling an application we can find an optimal number of wavefronts to execute
- Does a little better than CCWS.
- Limitations: Requires profiling, input dependent, does not exploit phase behavior.

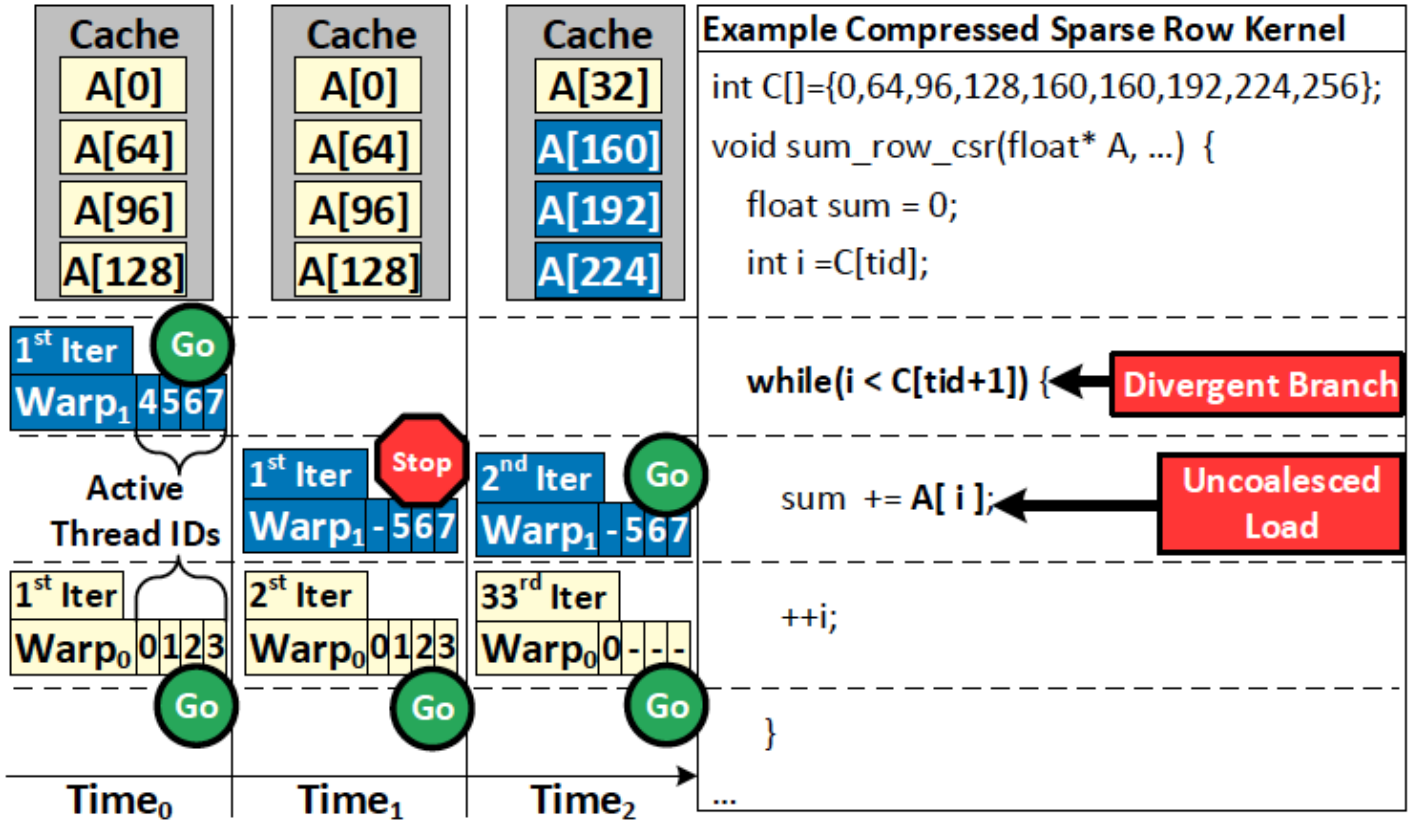
# Improve upon CCWS?

- CCWS detects bad scheduling decisions and avoids them in future.
- Would be better if we could “think ahead” / “be proactive” instead of “being reactive”

# Divergence Aware Warp Scheduling T. Rogers, M O'Conner, and T. Aamodt MICRO 2013 Goal

- Design a scheduler to match #scheduled wavefronts with the L1 cache size
  - Working set of the wavefronts fits in the cache
  - Emphasis on intra-wavefront locality
- Differs from CCWS in being proactive
  - Deeper look at what happens inside loops
  - Proactive
  - Explicitly Handles Divergence (both memory and control flow)

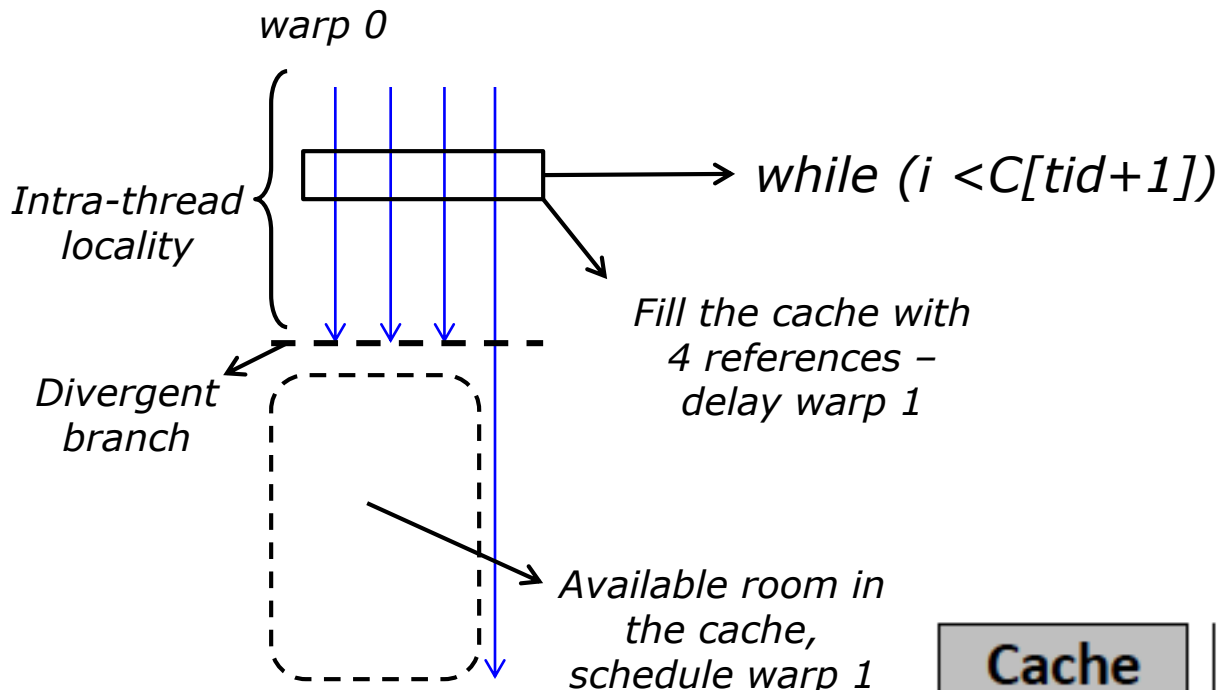
# Key Idea

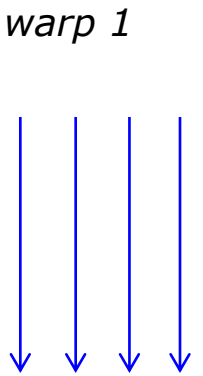



- Manage the relationship between control divergence, memory divergence and scheduling

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Key Idea (2)




Use warp 0 behavior to predict interference due to warp 1

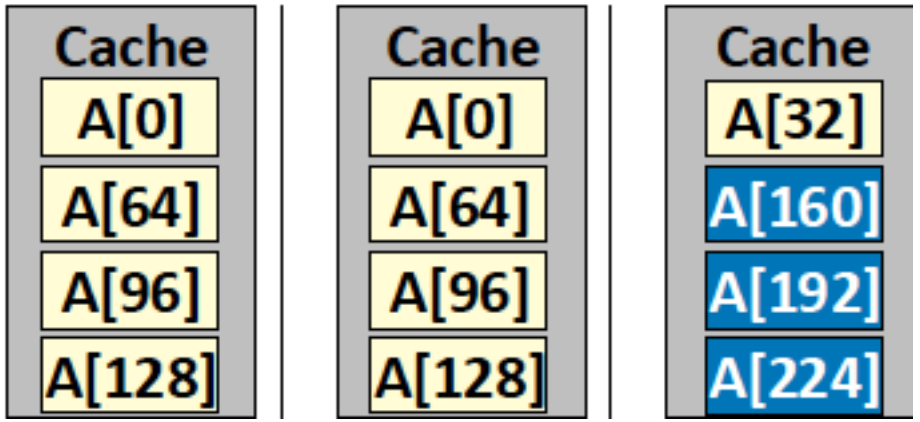



Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

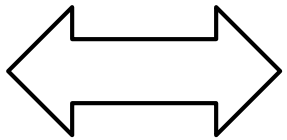
# Goal

## *Simpler portable version*

**Example 1** Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

*Make the performance equivalent*



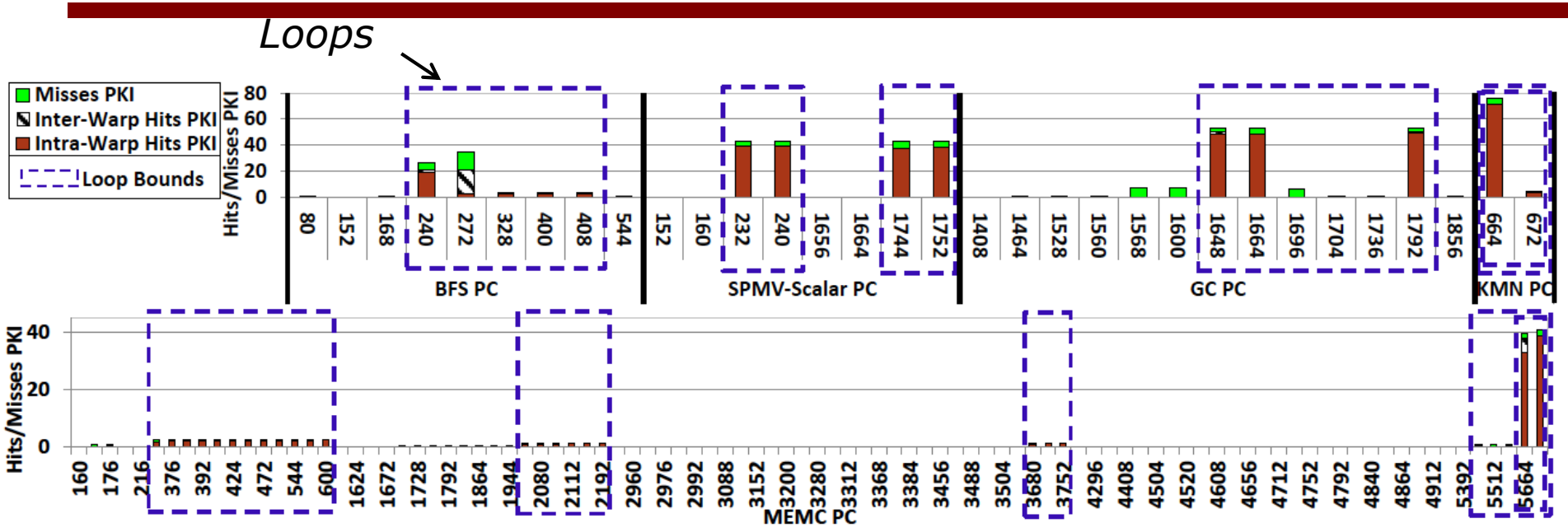
## *GPU-Optimized Version*

**Example 2** GPU-Optimized SPMV-Vector Kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;
    int myRow = (blockIdx.x * warpsPerBlock)
              + (t / warpSize);
    texReader vecTexReader;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

    if (myRow < dim)
    {
        int warpStart = rowDelimiters[myRow];
        int warpEnd = rowDelimiters[myRow+1];
        float mySum = 0;
        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
        if (id <  2)
            partialSums[t] += partialSums[t+ 2];
        if (id <  1)
            partialSums[t] += partialSums[t+ 1];

        // Write result
        if (id == 0)
        {
            out[myRow] = partialSums[t];
        }
    }
}
```

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Observation

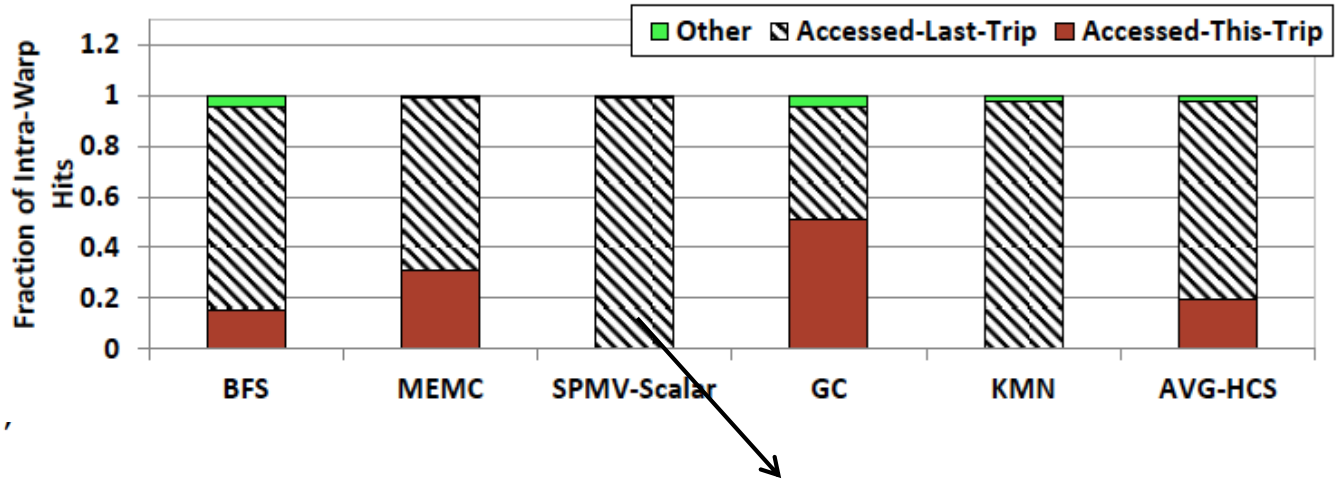



- Bulk of the accesses in a loop come from a few static load instructions
- Bulk of the locality in (these) applications is intra-loop

# Distribution of Locality

**Example 1** Highly Divergent SPM

```
__global__ void
spmv_csr_scalar_kernel(cor
                       cor
                       cor
                       cor
                       flo
{
    int myRow = blockIdx.x
        + threadIdx.x;
    texReader vecTexReader,

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```




*Hint: Can we keep data from last iteration?*

Bulk of the locality comes form a few static loads in loops

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# A Solution

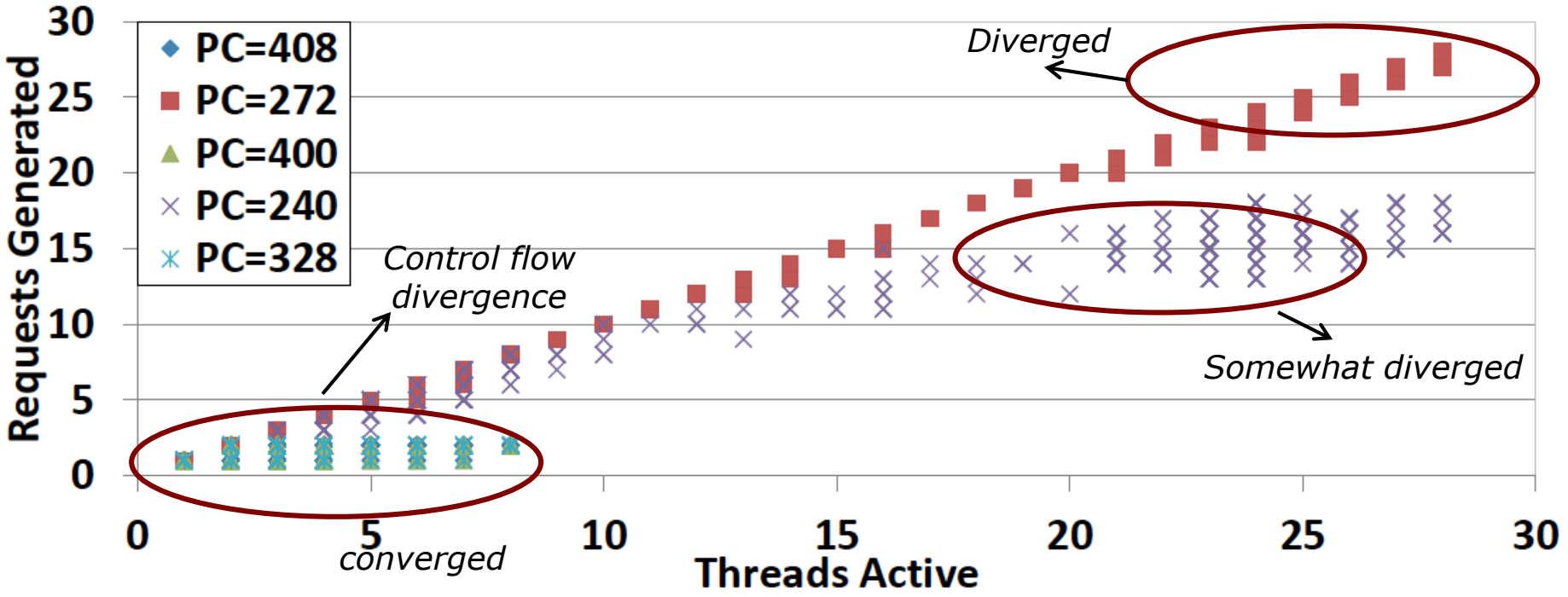
**Example 1** Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

- Prediction mechanisms for locality across iterations of a loop
- Schedule such that data fetched in one iteration is still present at next iteration
- Combine with control flow divergence (how much of the footprint needs to be in the cache?)

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

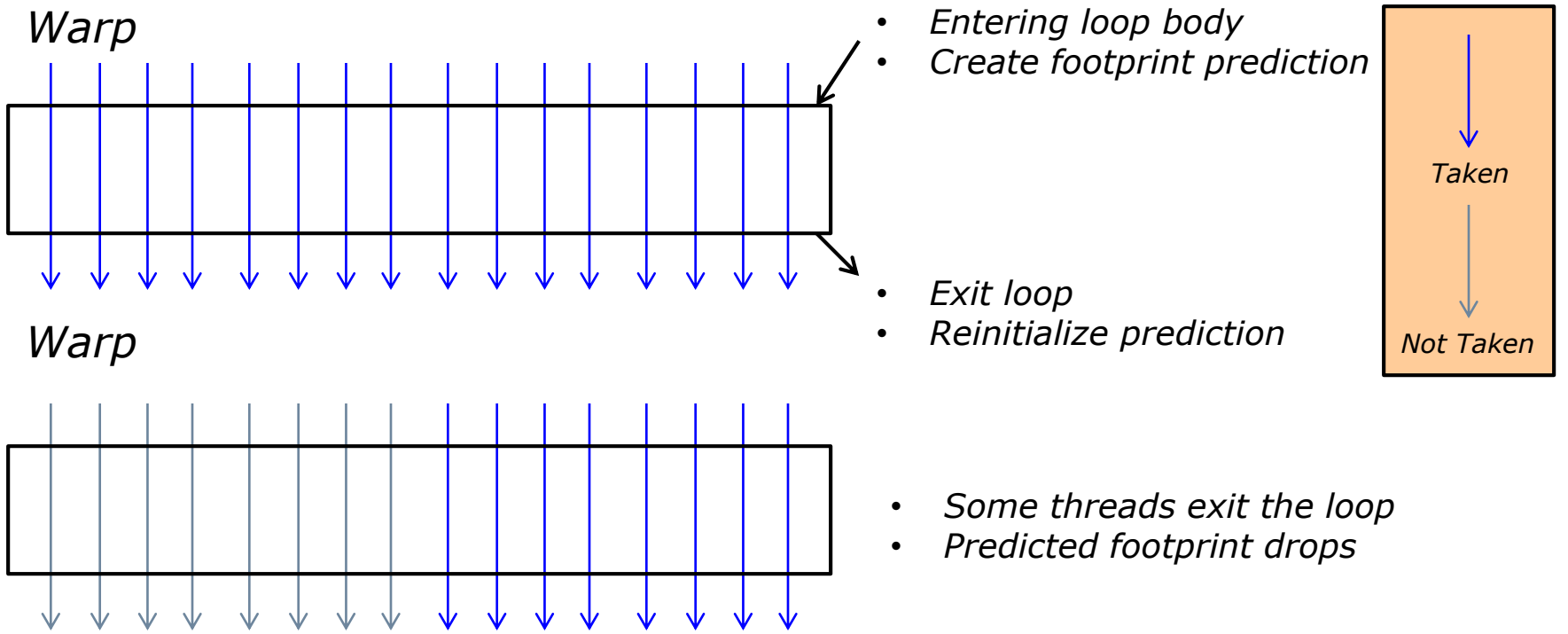
# Classification of Dynamic Loads




- Group static loads into equivalence classes → reference the same cache line
- Identify these groups by repetition ID
- Prediction for each load by compiler or hardware

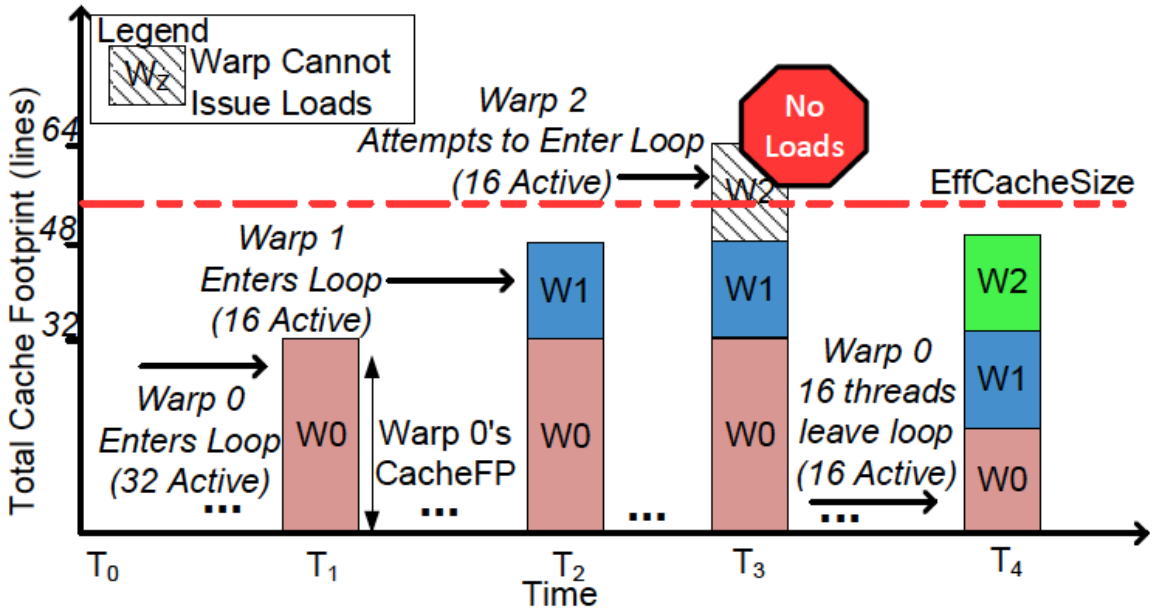
Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Predicting a Warp's Cache Footprint




- Predict locality usage of static loads
  - Not all loads increase the footprint
- Combine with control divergence to predict footprint
- Use footprint to throttle/not-throttle warp issue

# Principles of Operation




- Prefix sum of each warp's cache footprint used to select warps that can be issued

$$EffCacheSize = kAssocFactor.TotalNumLines$$

  - *Scaling back from a fully associative cache*
  - *Empirically determined*

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Principles of Operation (2)

**Example 1** Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

- Profile static load instructions
  - Are they divergent?
  - Loop repetition ID
    - Assume all loads with same base address and offset within cache line access are repeated each iteration

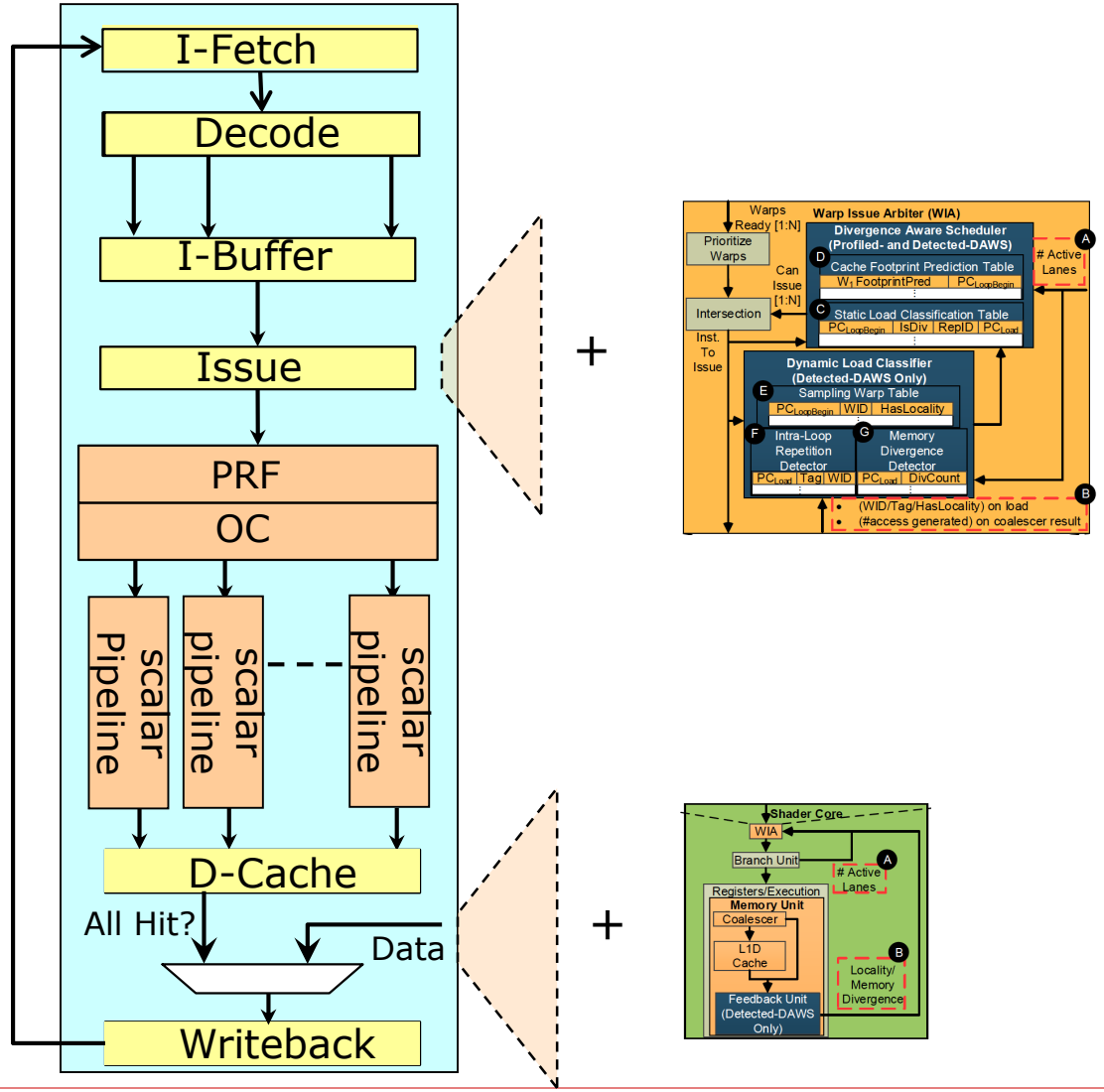
Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Prediction Mechanisms

- Profiled Divergence Aware Scheduling (DAWS)
  - Used offline profile results to dynamically determine de-scheduling decisions

- Detected Divergence Aware Scheduling (DAWS)
  - Behaviors derived at run-time to drive de-scheduling decisions
    - Loops that exhibit intra-warp locality
    - Static loads are characterized as divergent or convergent

# Extensions for DAWS




+




+

# Operation: Tracking

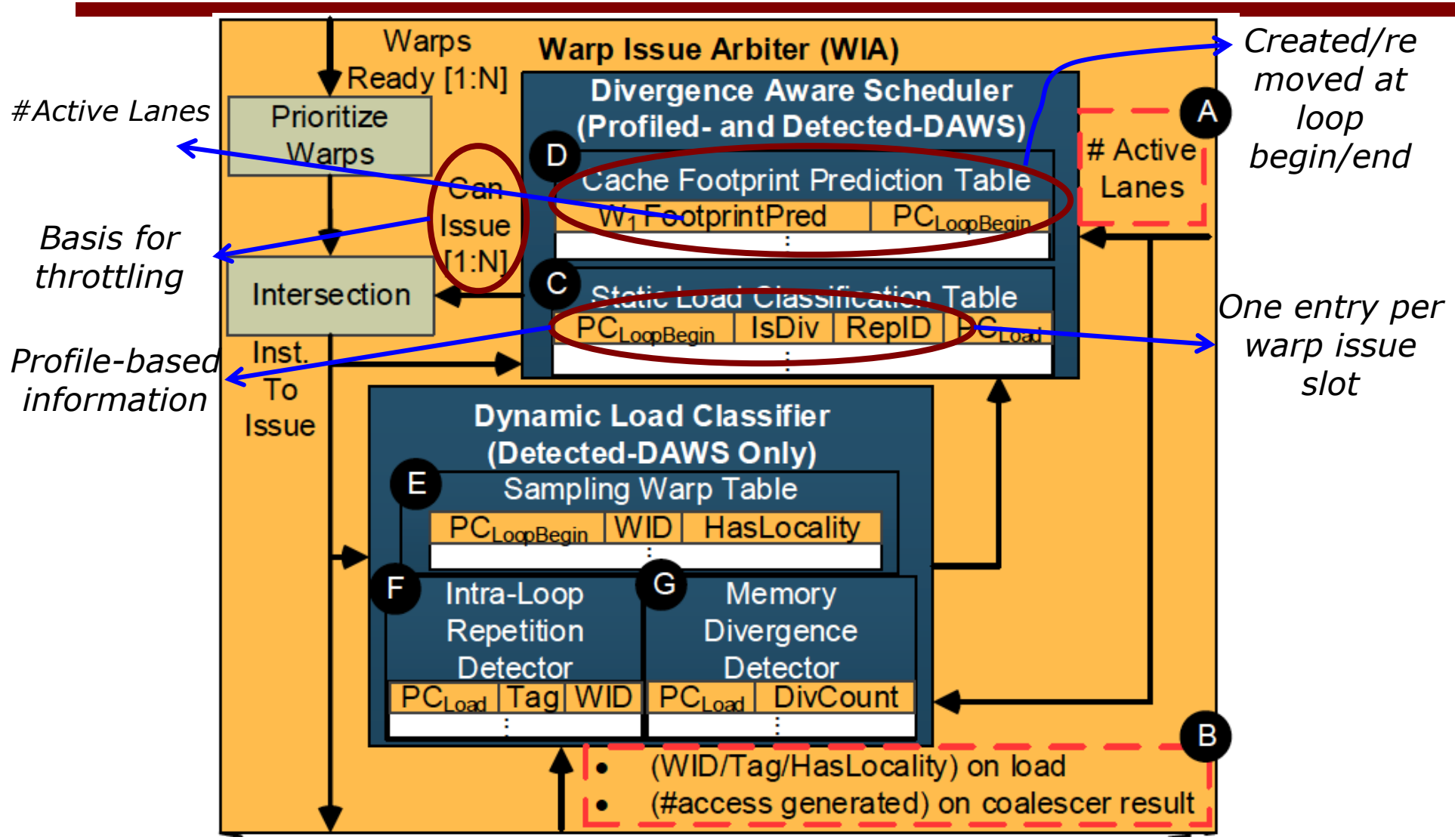
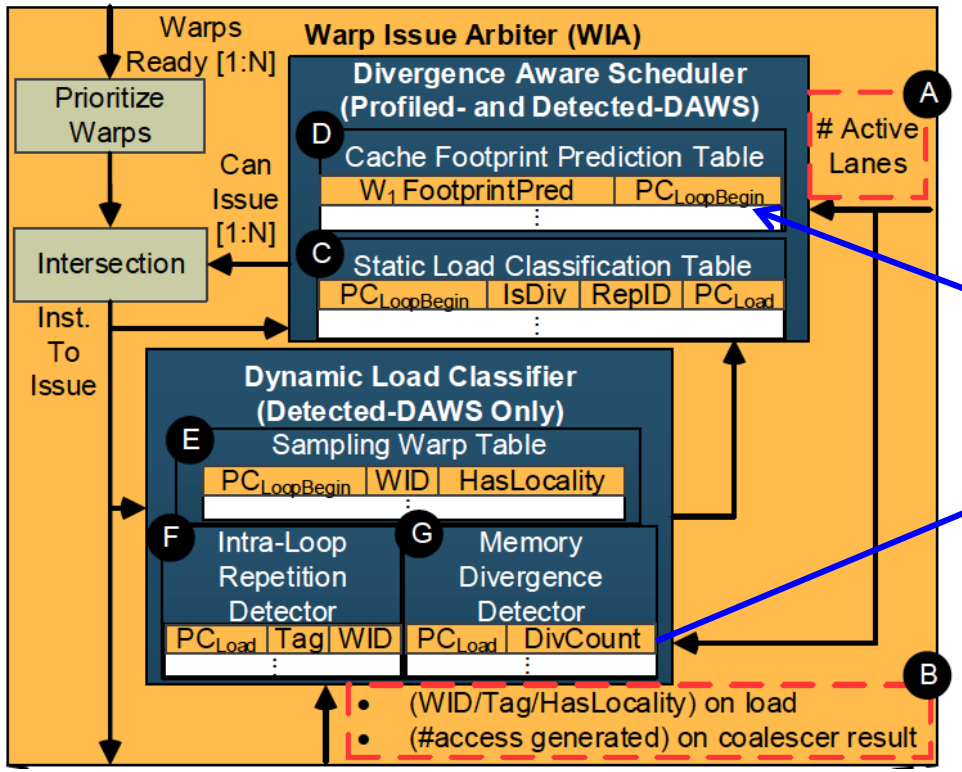



Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Operation: Prediction



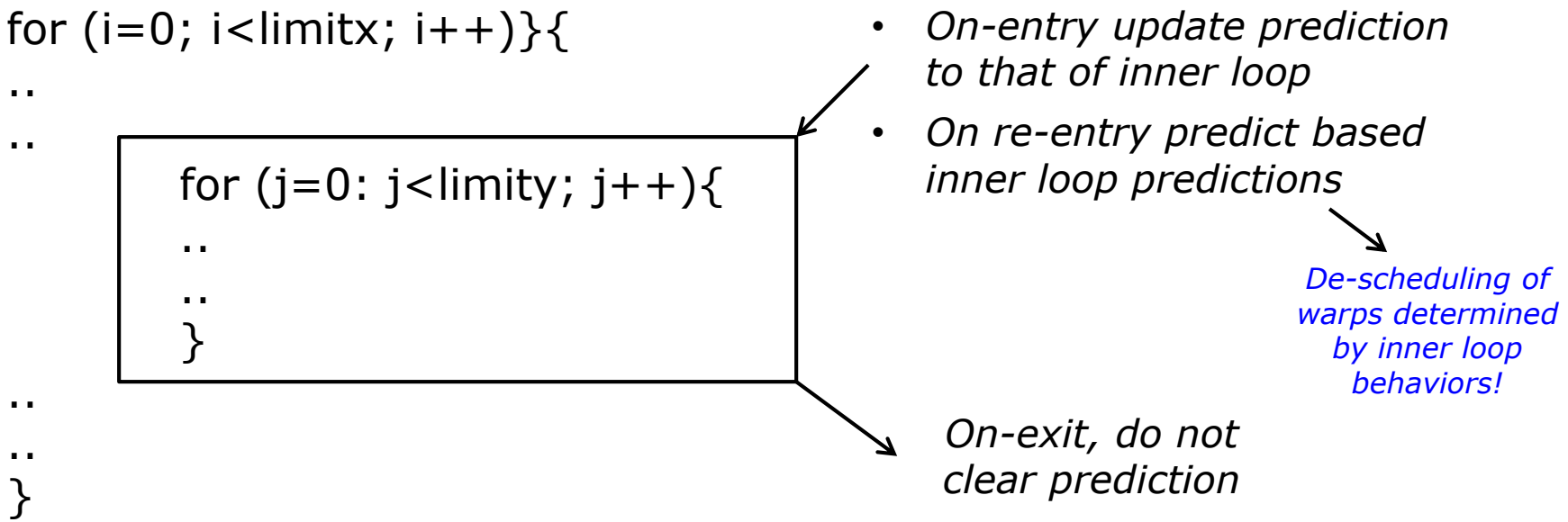

*Sum results from static loads in this loop*

- *Add #active-lanes of cache lines for divergent loads*
- *Add 2 for converged loads*
- *Count loads in the same equivalence class only once (unless divergent)*

- Generally only considering de-scheduling warps in loops
  - Since most of the activity is here
- Can be extended to non-loop regions by associating non-loop code with next loop

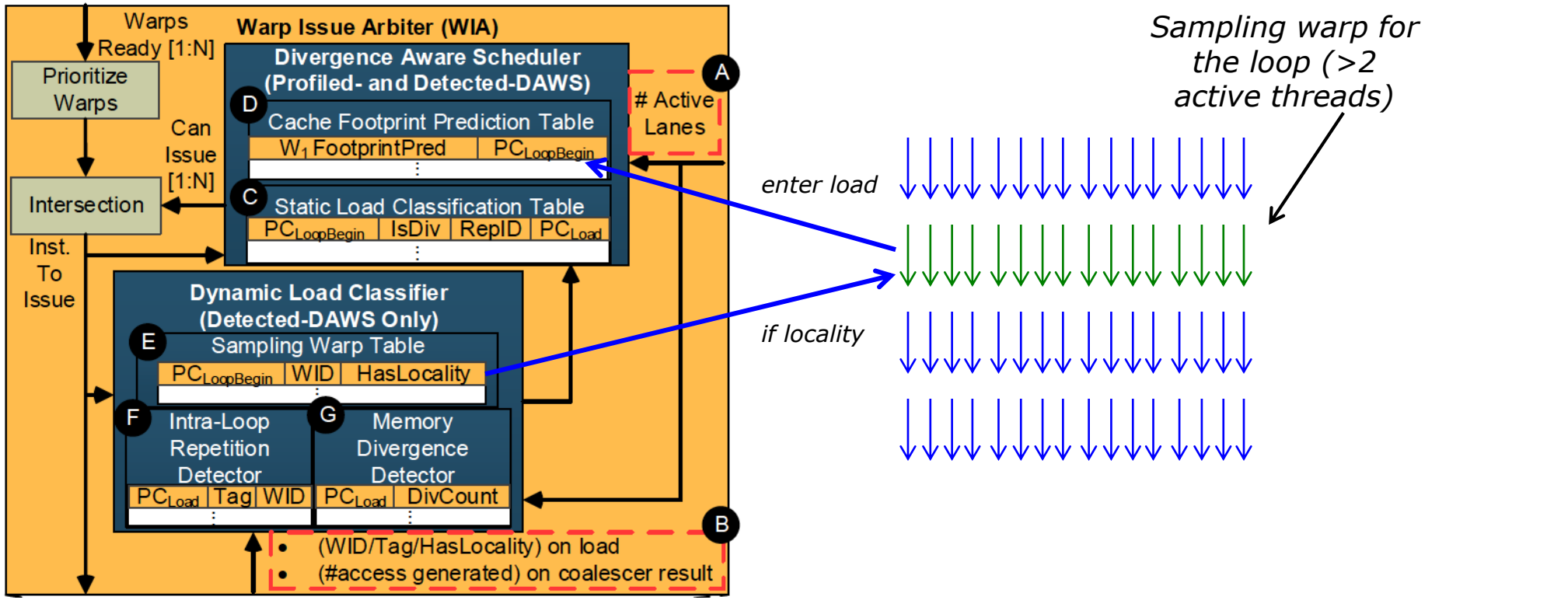
Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Operation: Nested Loops




- Re-used predictions based on inner-most loops which is where most of the date re-use is found

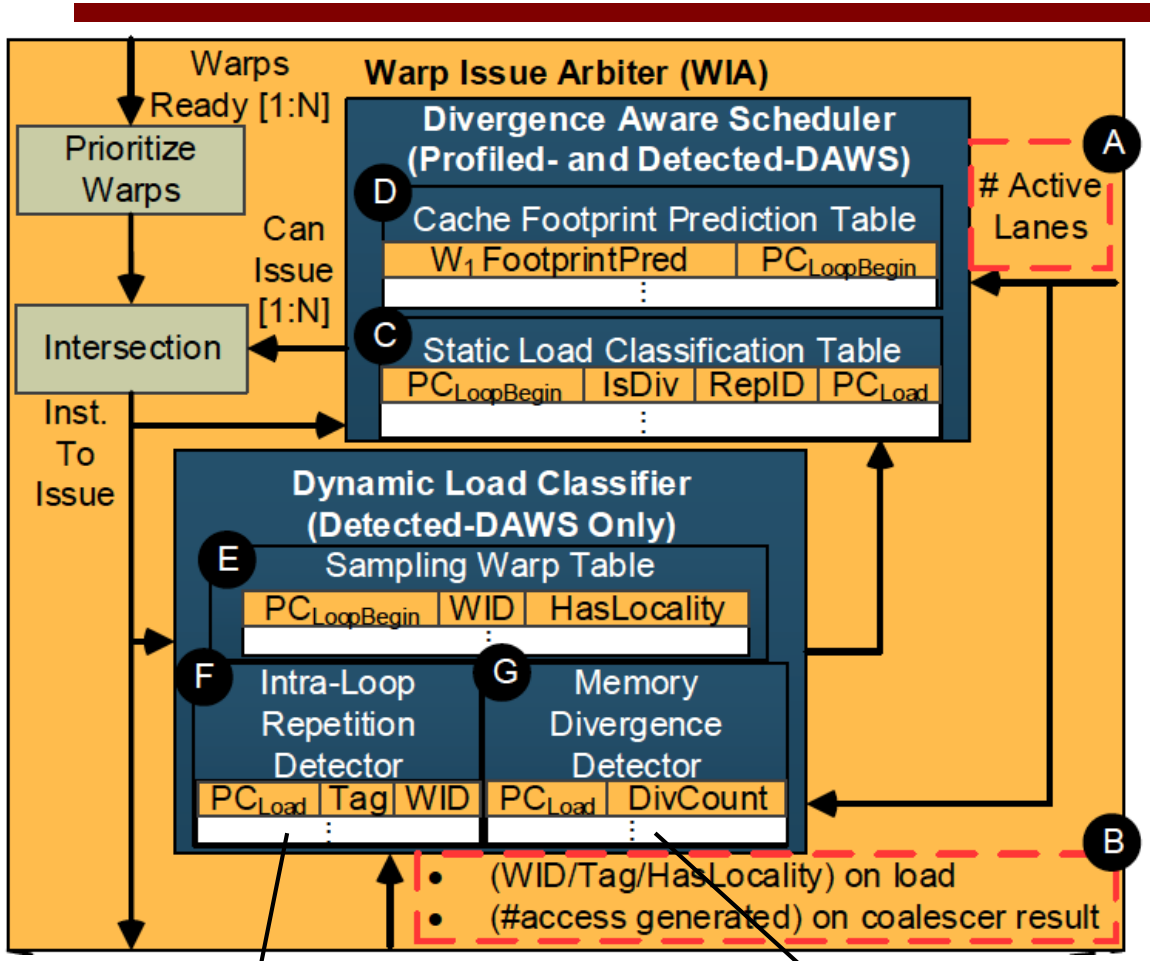
# Detected DAWS: Prediction

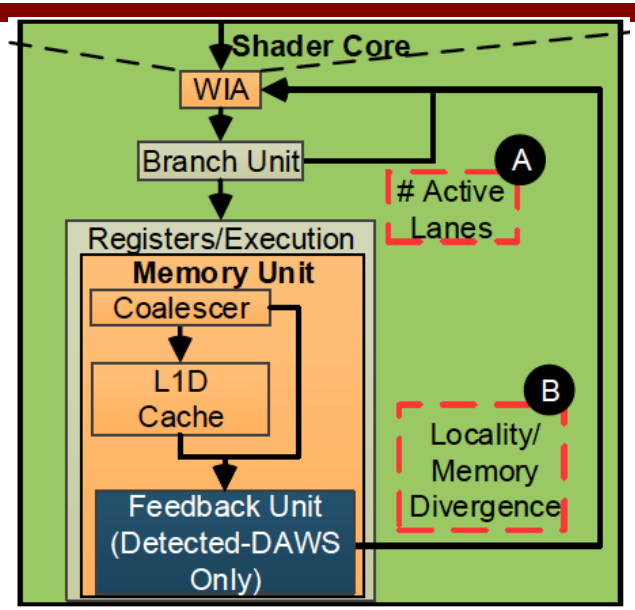



- Detect both memory divergence and intra-loop repetition at run time
- Fill $PC_{Load}$ entries based on run time information

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Detected DAWS: Classification




*Create equivalence classes of loads (checking the PCs)*

*Increment or decrement the counter depending on #memory accesses for a load*

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Performance

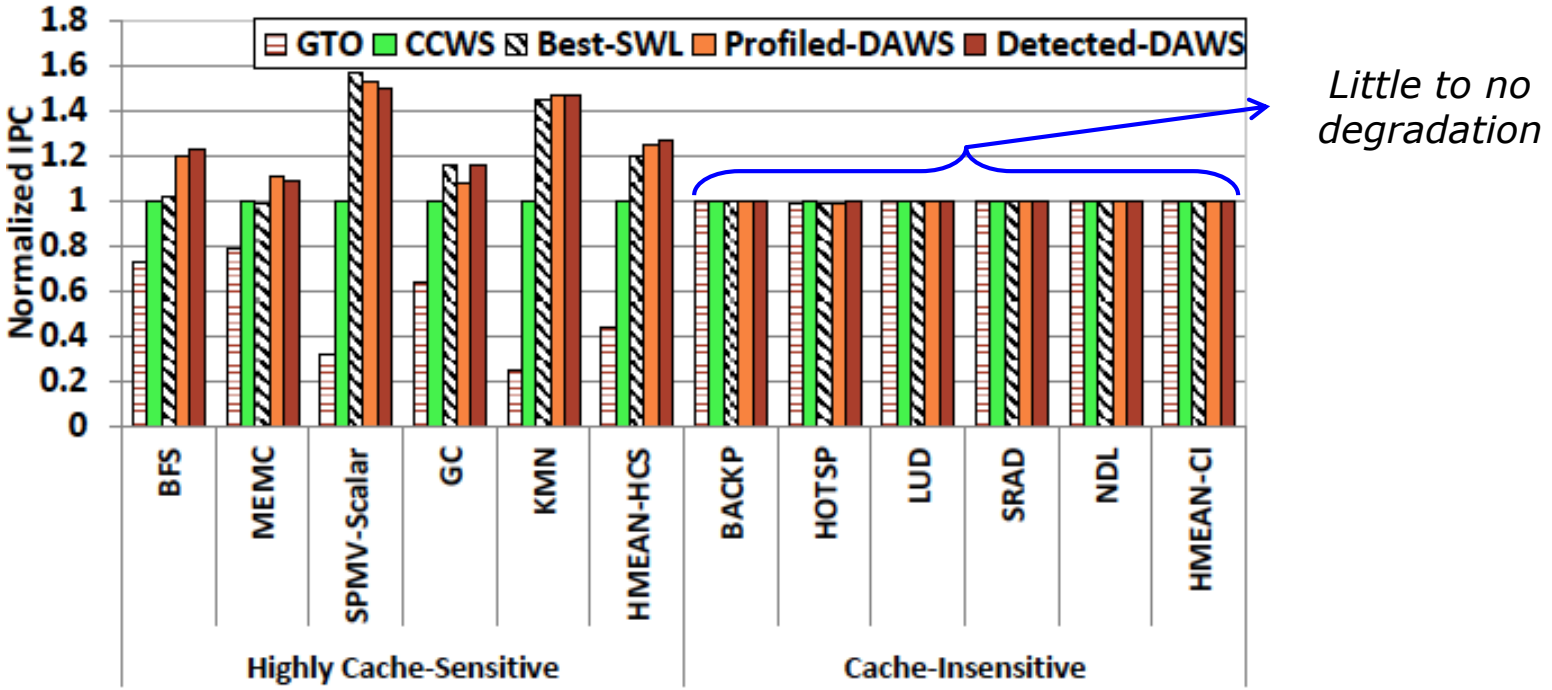
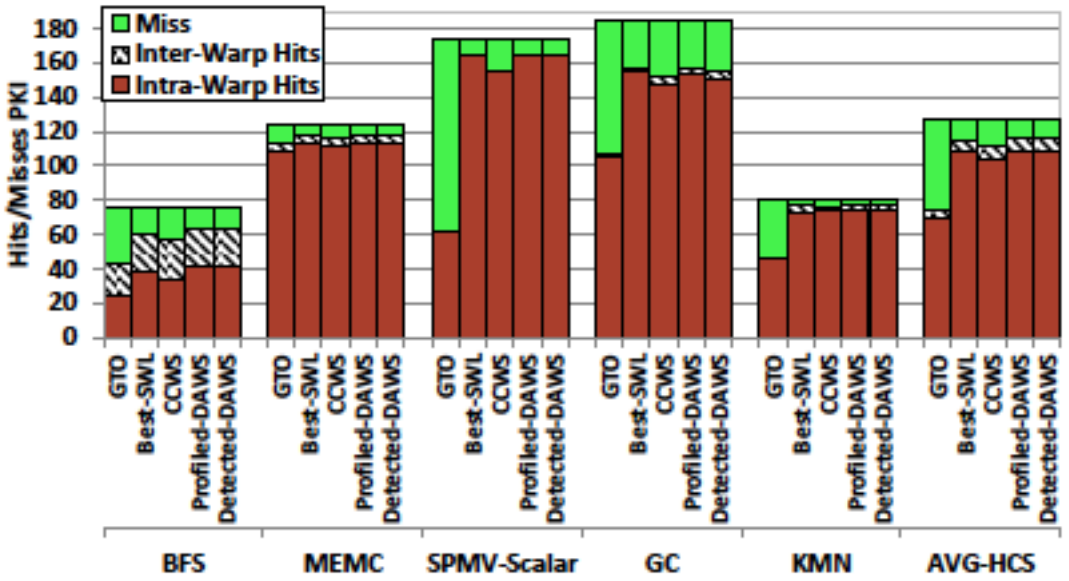
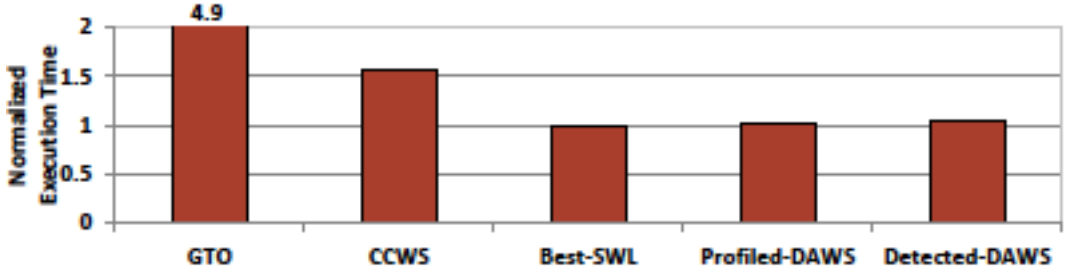



Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

# Performance




*Significant intra-warp locality*




*SPMV-scalar normalized to best SPMV-vector*

Figure from T. Rogers, M. O/Connor, T. Aamodt, "Divergence-Aware Warp Scheduling," MICRO 2013

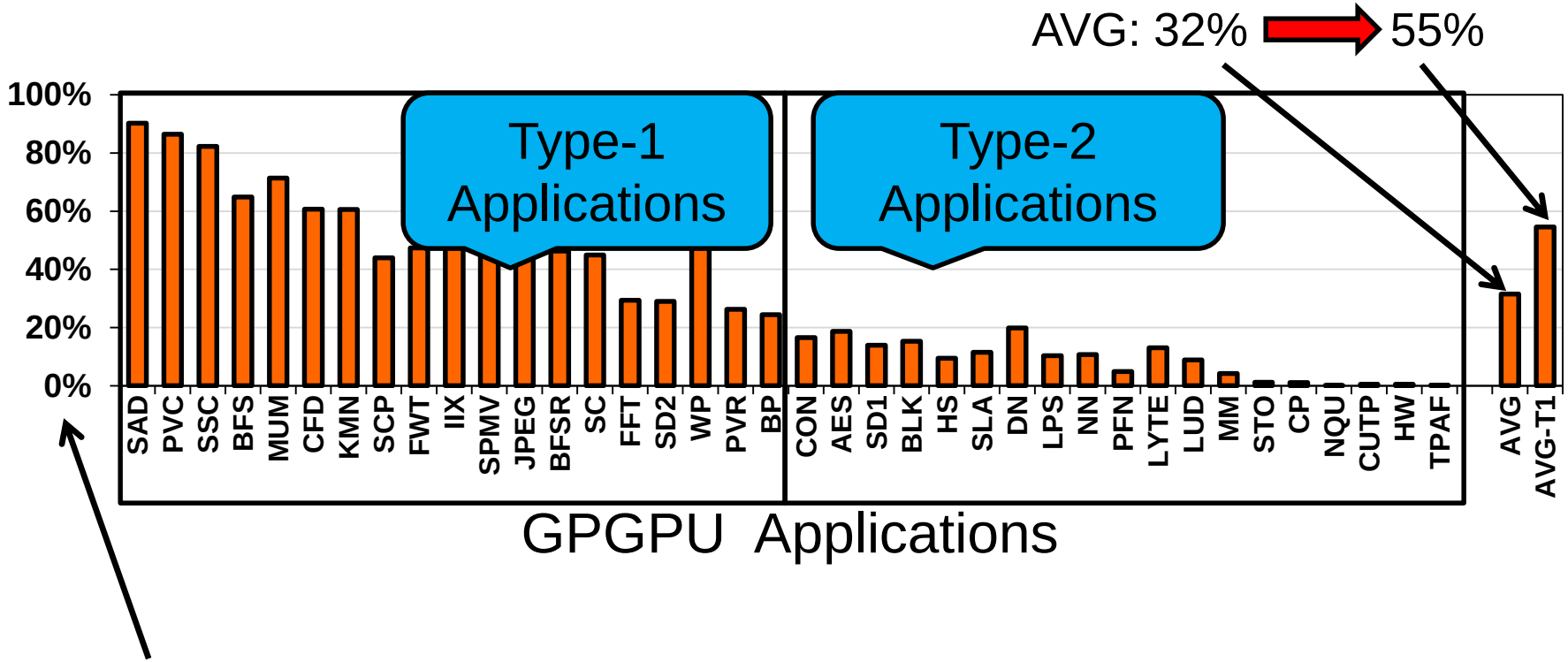
# Summary

- If we can characterize warp level memory reference locality, we can use this information to minimize interference in the cache through scheduling constraints
- Proactive scheme outperforms reactive management
- Understand interactions between memory divergence and control divergence

# OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance
# A. Jog et. al ASPLOS 2013 Goal

- Understand memory effects of scheduling from deeper within the memory hierarchy
- Minimize idle cycles induced by stalling warps waiting on memory references

# Off-chip Bandwidth is Critical!




Percentage of total execution cycles wasted waiting for the data to come back from DRAM

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Source of Idle Cycles

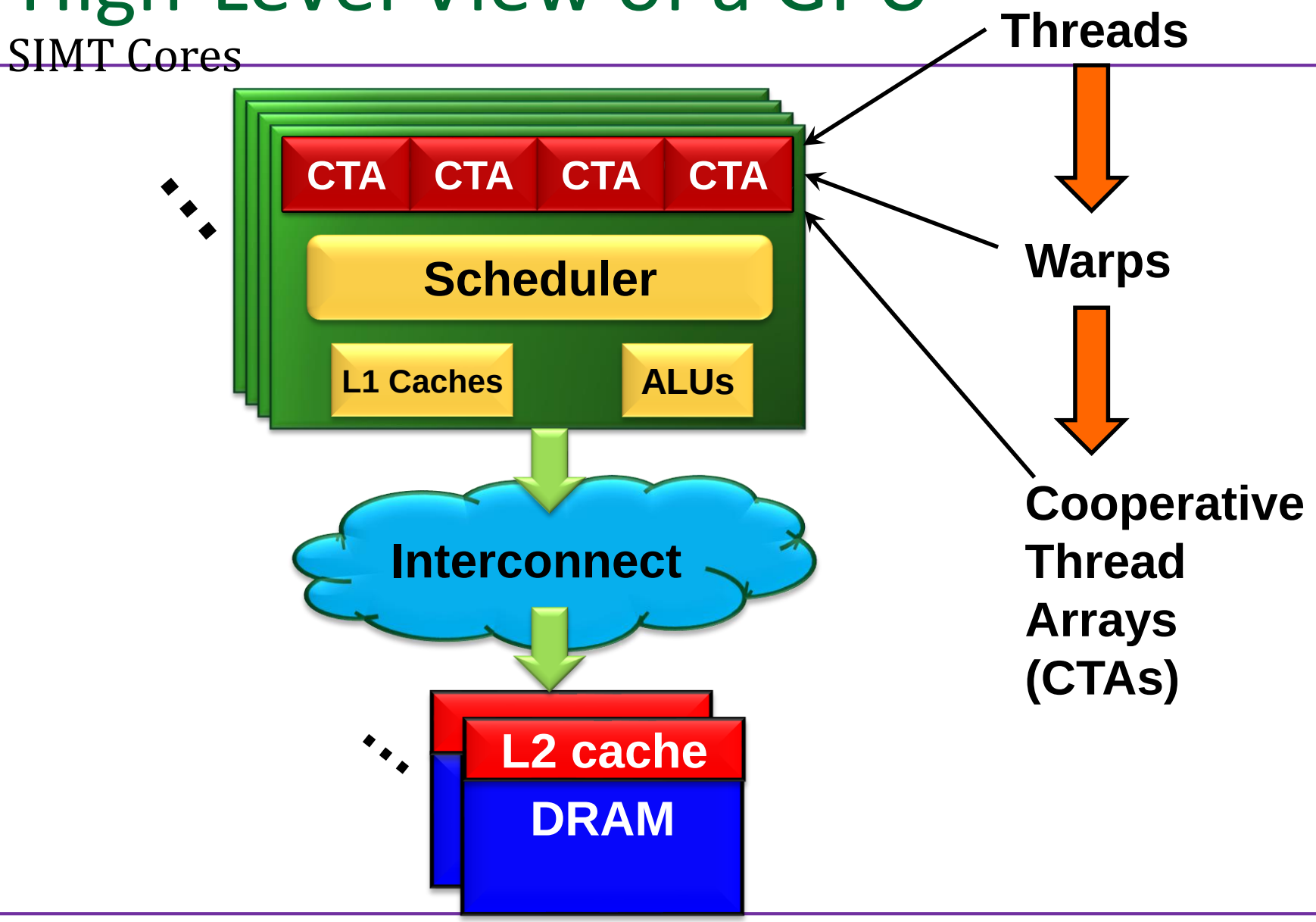
- Warps stalled on waiting for memory reference
  - Cache miss
  - Service at the memory controller
  - Row buffer miss in DRAM
  - Latency in the network (not addressed in this paper)
- The last warp effect
- The last CTA effect
- Lack of multiprogrammed execution
  - One (small) kernel at a time

# Impact of Idle Cycles

| # | App. Suite | **Type-1 Applications** | Abbr. | PMEM | CINV |
|---|---|---|---|---|---|
| 1 | Parboil | Sum of Abs. Differences | SAD | H (6.39x) | 91% |
| 2 | MapReduce | PageViewCount | PVC | H (4.99x) | 93% |
| 3 | MapReduce | SimilarityScore | SSC | H (4.60x) | 85% |
| 4 | CUDA SDK | Breadth First Search | BFS | H (2.77x) | 81% |
| 5 | CUDA SDK | MUMerGPU | MUM | H (2.66x) | 72% |
| 6 | Rodinia | CFD Solver | CFD | H (2.46x) | 66% |
| 7 | Rodinia | Kmeans Clustering | KMN | H (2.43x) | 65% |
| 8 | CUDA SDK | Scalar Product | SCP | H (2.37x) | 58% |
| 9 | CUDA SDK | Fast Walsh Transform | FWT | H (2.29x) | 58% |
| 10 | MapReduce | InvertedIndex | IIX | H (2.29x) | 65% |
| 11 | Parboil | Sparse-Matrix-Mul. | SPMV | H (2.19x) | 65% |
| 12 | 3rd Party | JPEG Decoding | JPEG | H (2.12x) | 54% |
| 13 | Rodinia | Breadth First Search | BFSR | H (2.09x) | 64% |
| 14 | Rodinia | Streamcluster | SC | H (1.94x) | 52% |
| 15 | Parboil | FFT Algorithm | FFT | H (1.56x) | 37% |
| 16 | Rodinia | SRAD2 | SD2 | H (1.53x) | 36% |
| 17 | CUDA SDK | Weather Prediction | WP | H (1.50x) | 54% |

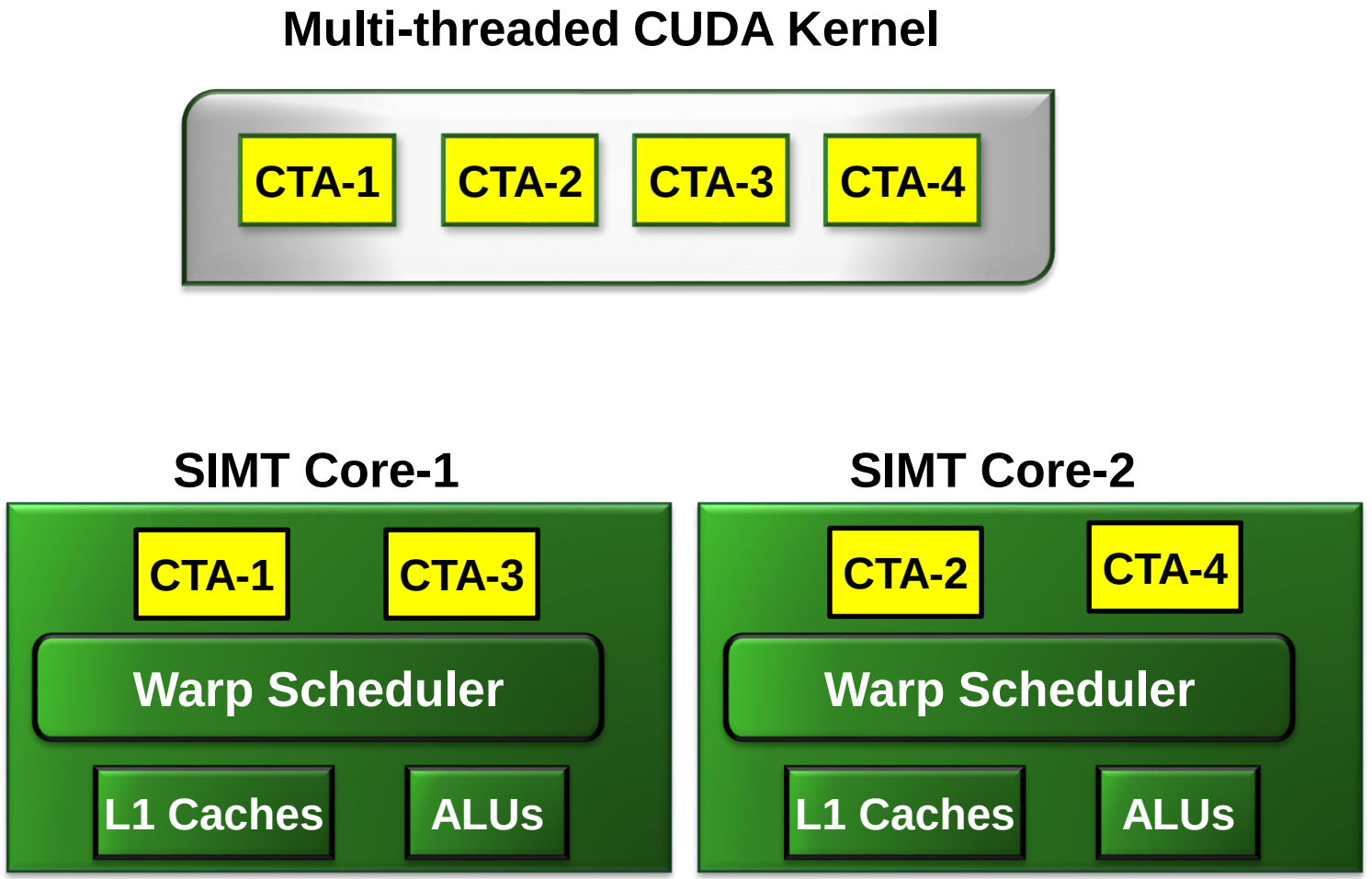
Figure from A. Jog et.al, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# High-Level View of a GPU




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# CTA-Assignment Policy (Example)




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Organizing CTAs Into Groups

- Set minimum number of warps equal to #pipeline stages
  - Same philosophy as the two-level warp scheduler
- Use same CTA grouping/numbering across SMs?

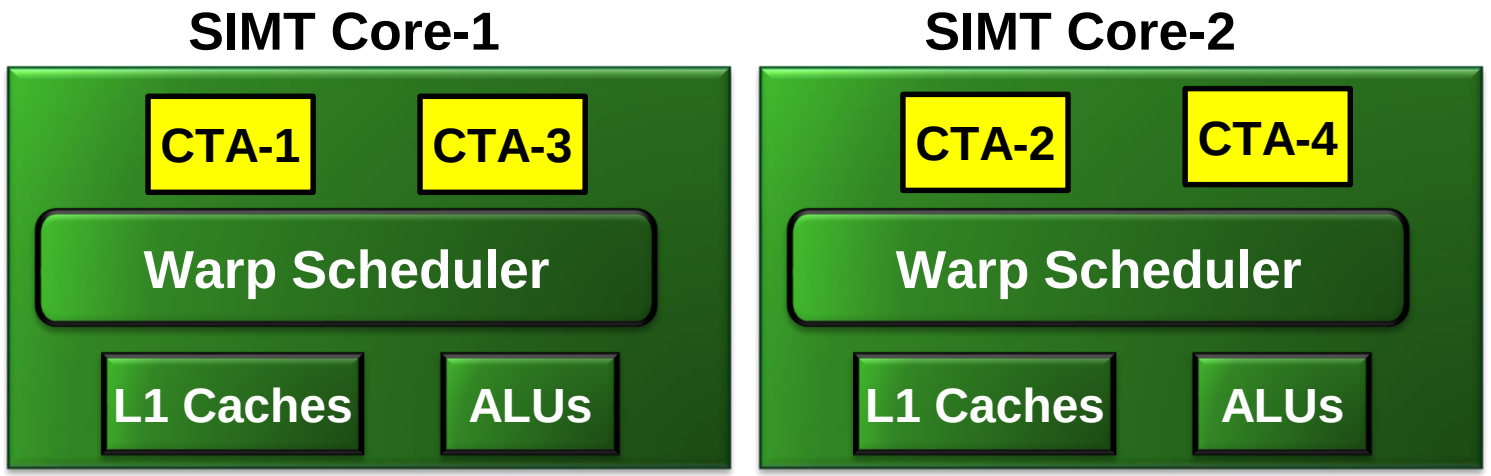
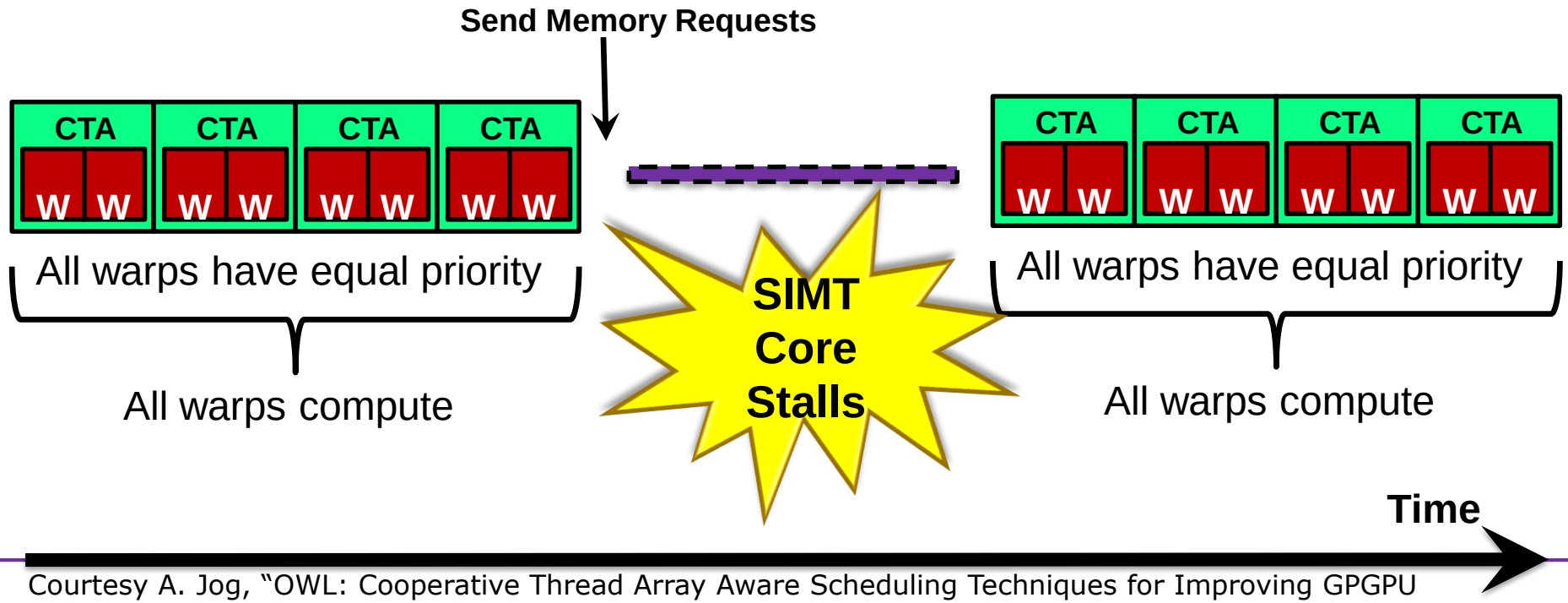



Figure from A. Jog et.al, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Warp Scheduling Policy

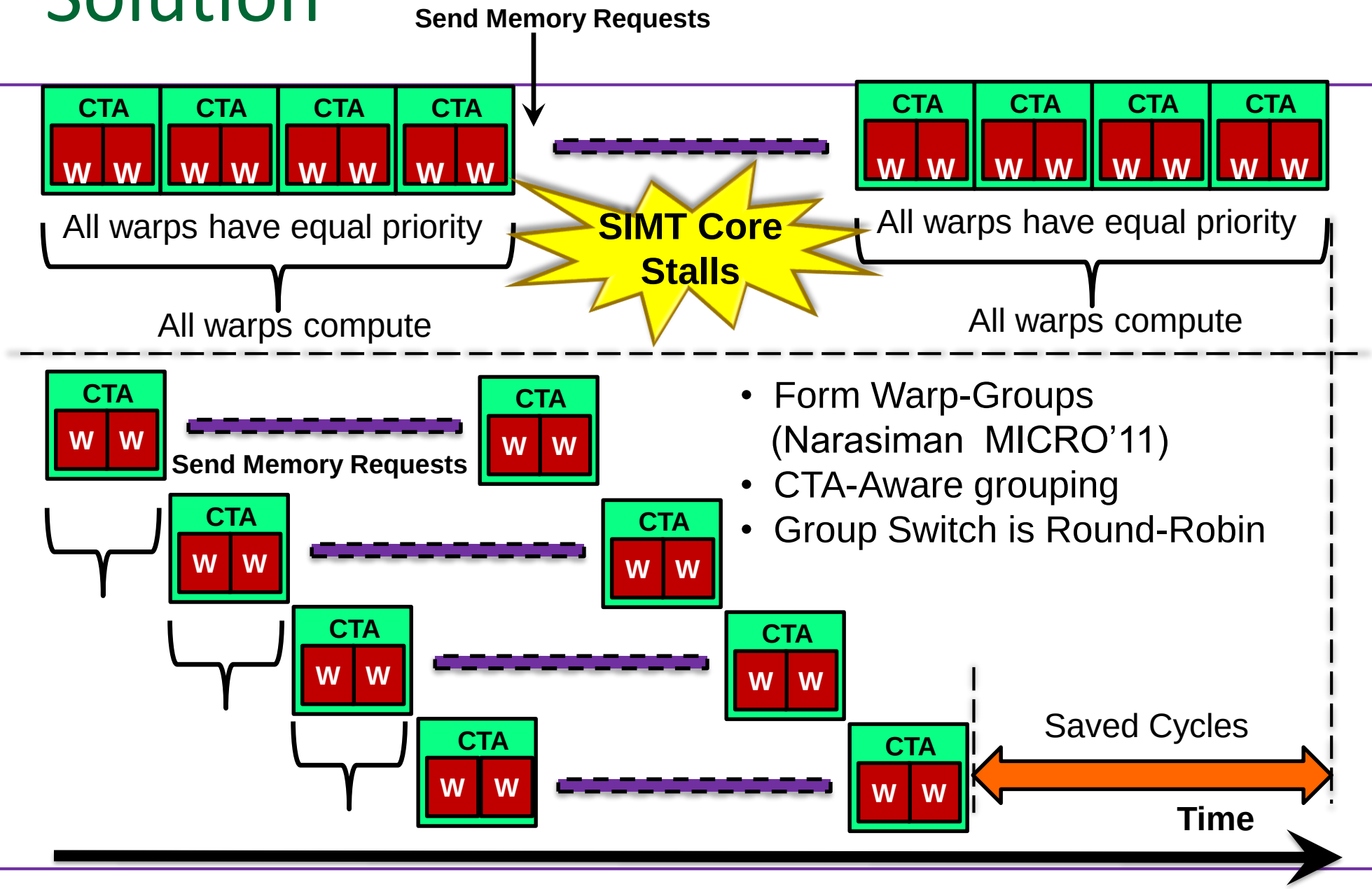
- All launched warps on a SIMT core have equal priority
  - Round-Robin execution

- Problem: Many warps stall at long latency operations roughly at the same time



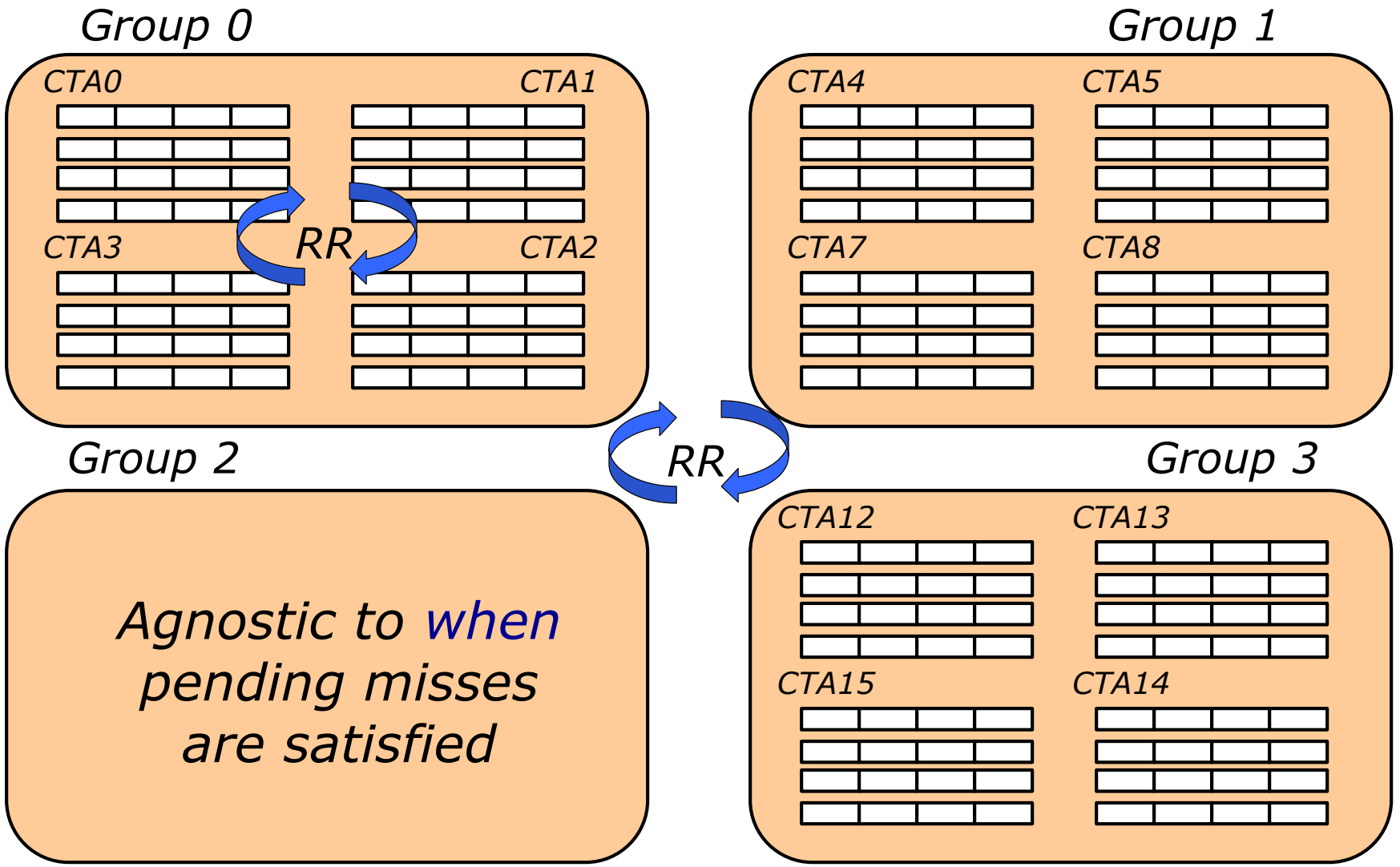

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Solution



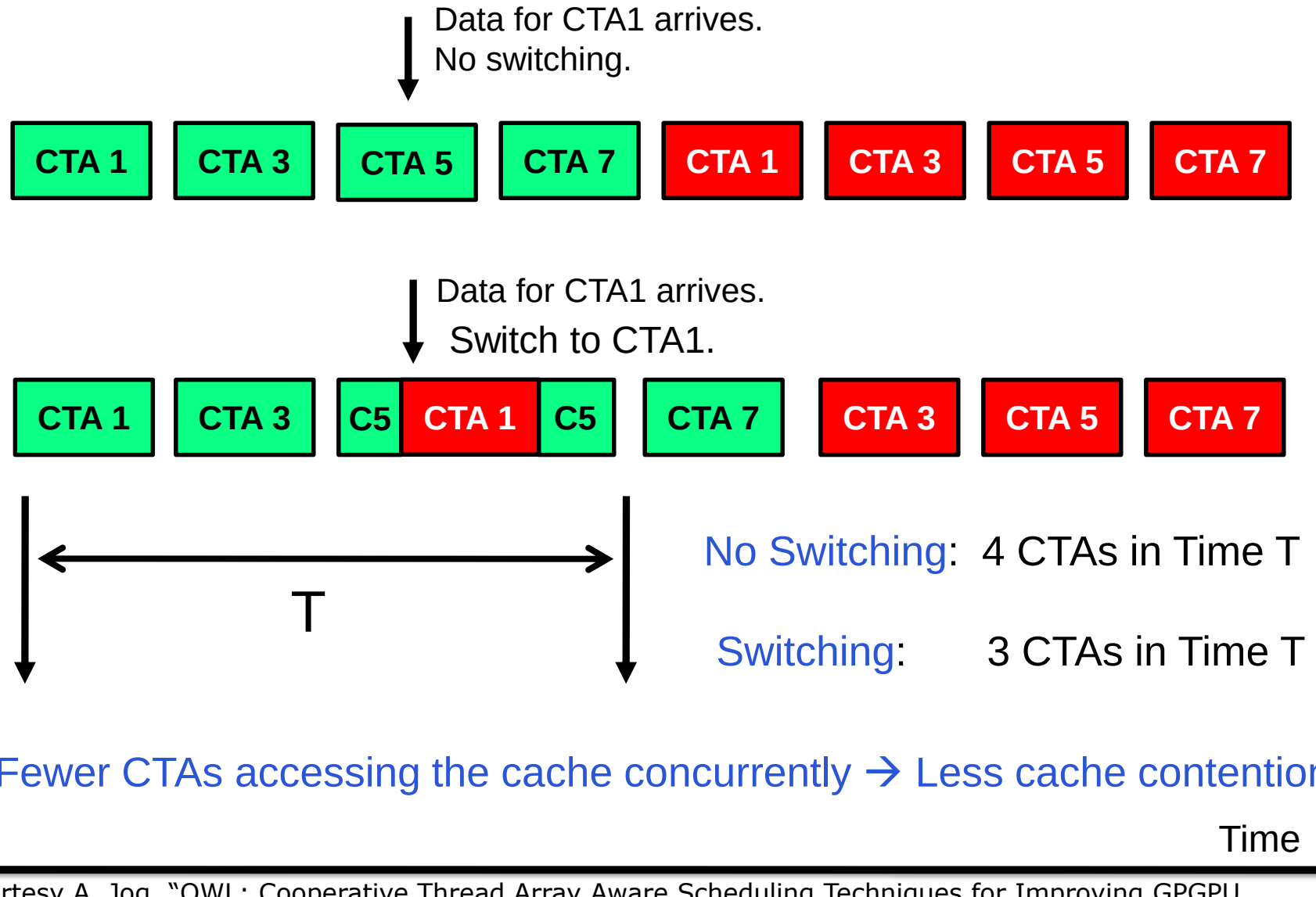

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Two Level Round Robin Scheduler



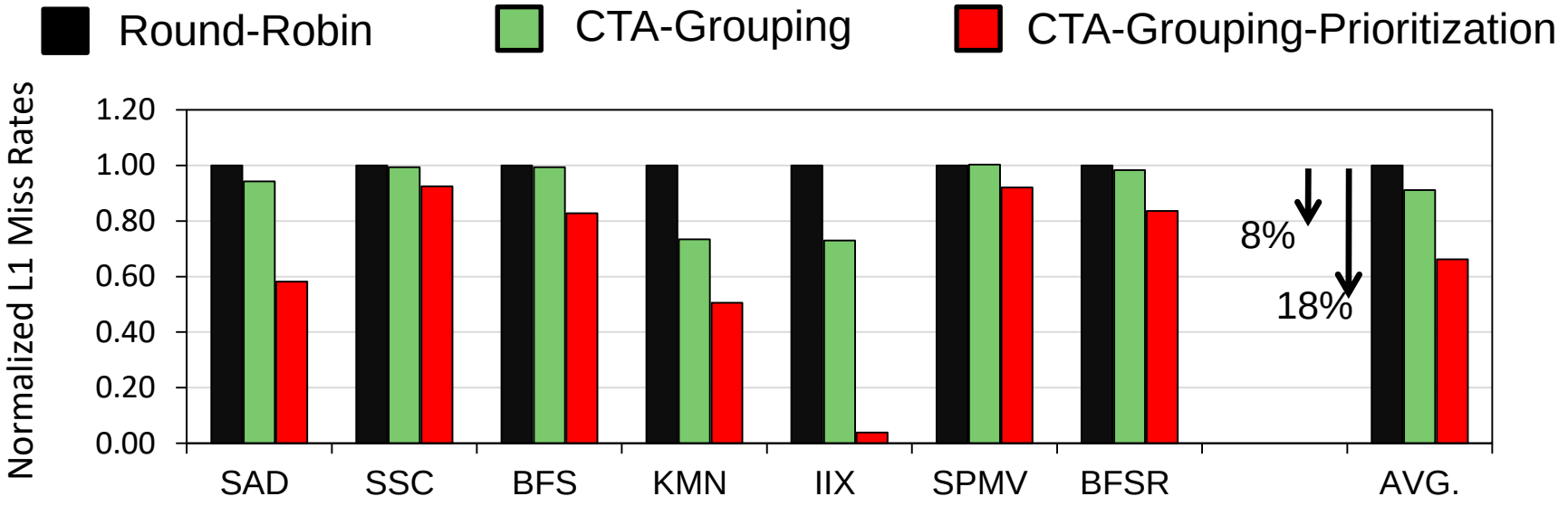

Thread

# Objective 1: Improve Cache Hit Rates




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

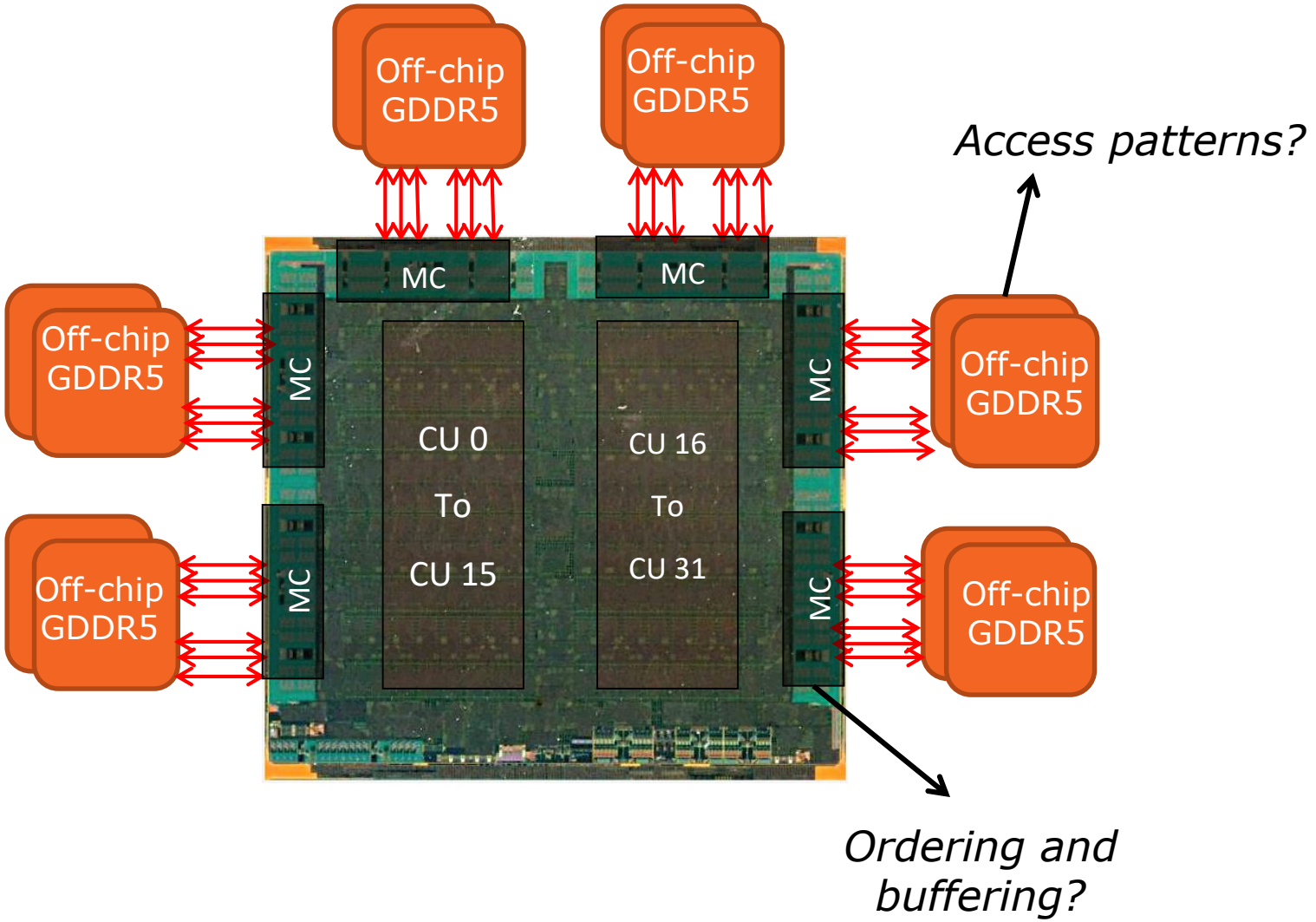
# Reduction in L1 Miss Rates




- Limited benefits for cache insensitive applications
- What is happening deeper in the memory system?

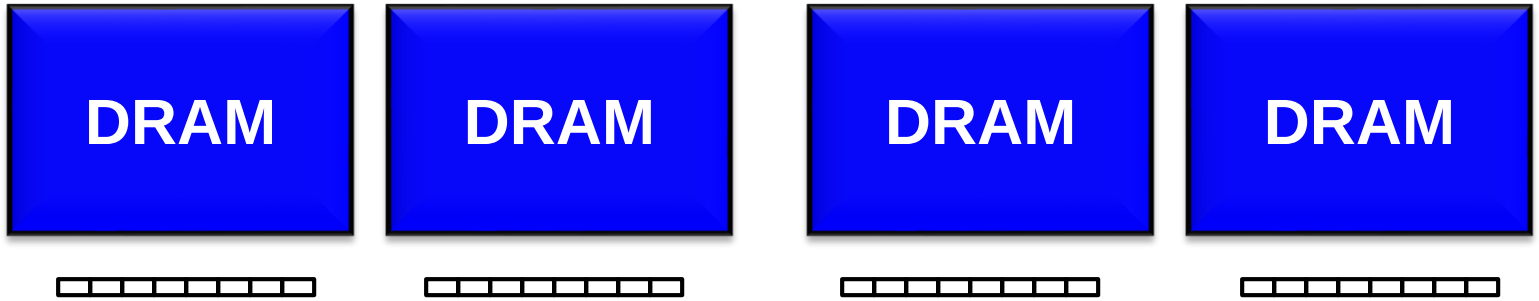
Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# The Off-Chip Memory Path

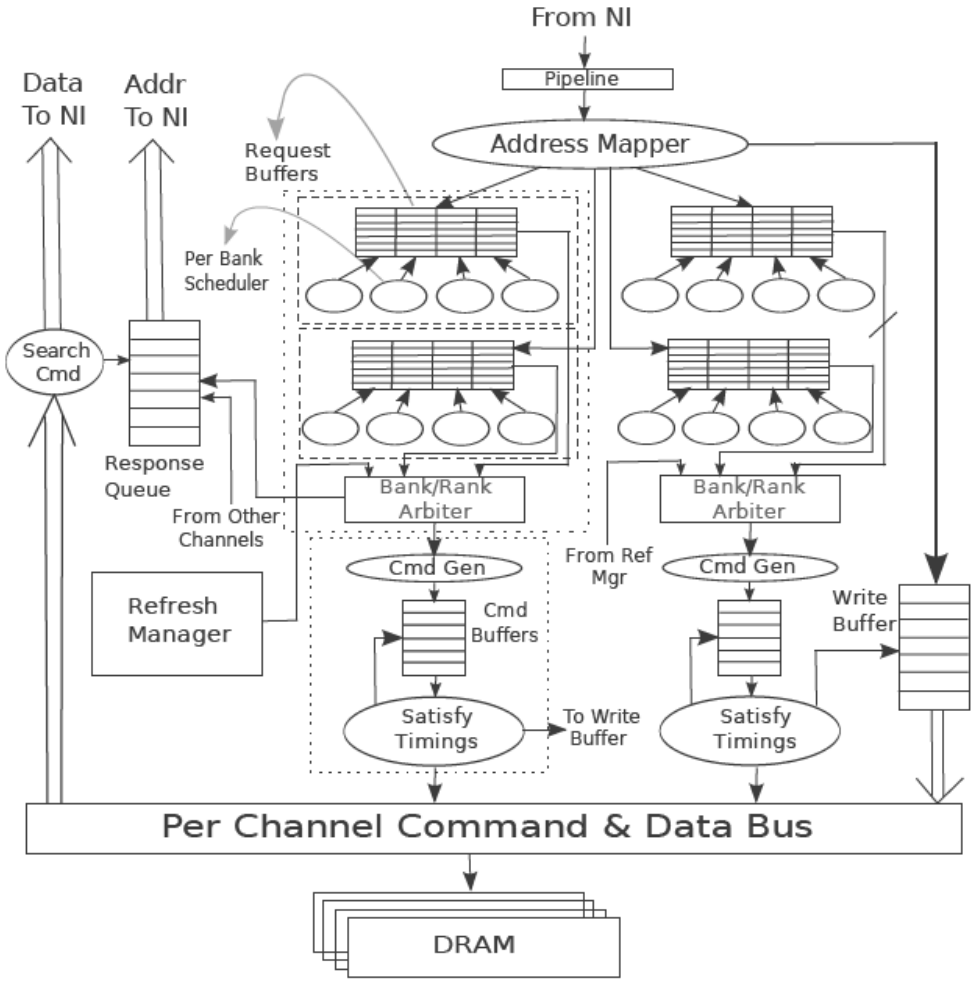


# Inter-CTA Locality


*How do CTAs Interact at the MC and in DRAM?*

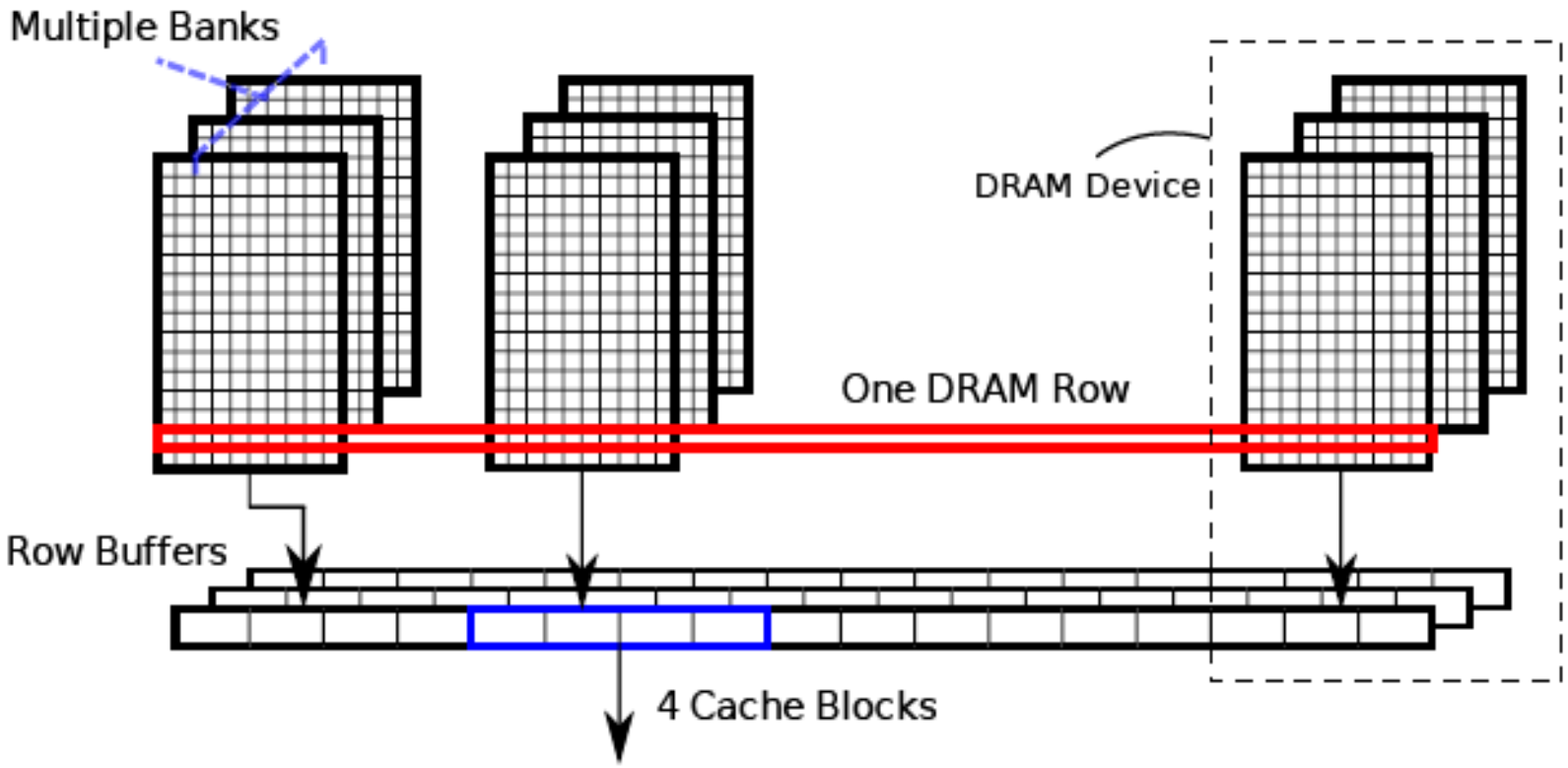


# Impact of the Memory Controller




- Memory scheduling policies
  - Optimize BW vs. memory latency
- Impact of row buffer access locality
- Cache lines?

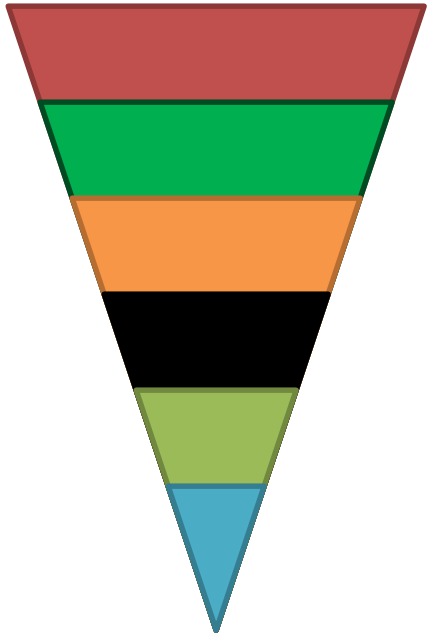
# Row Buffer Locality



# The DRAM Subsystem

# DRAM Subsystem Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column

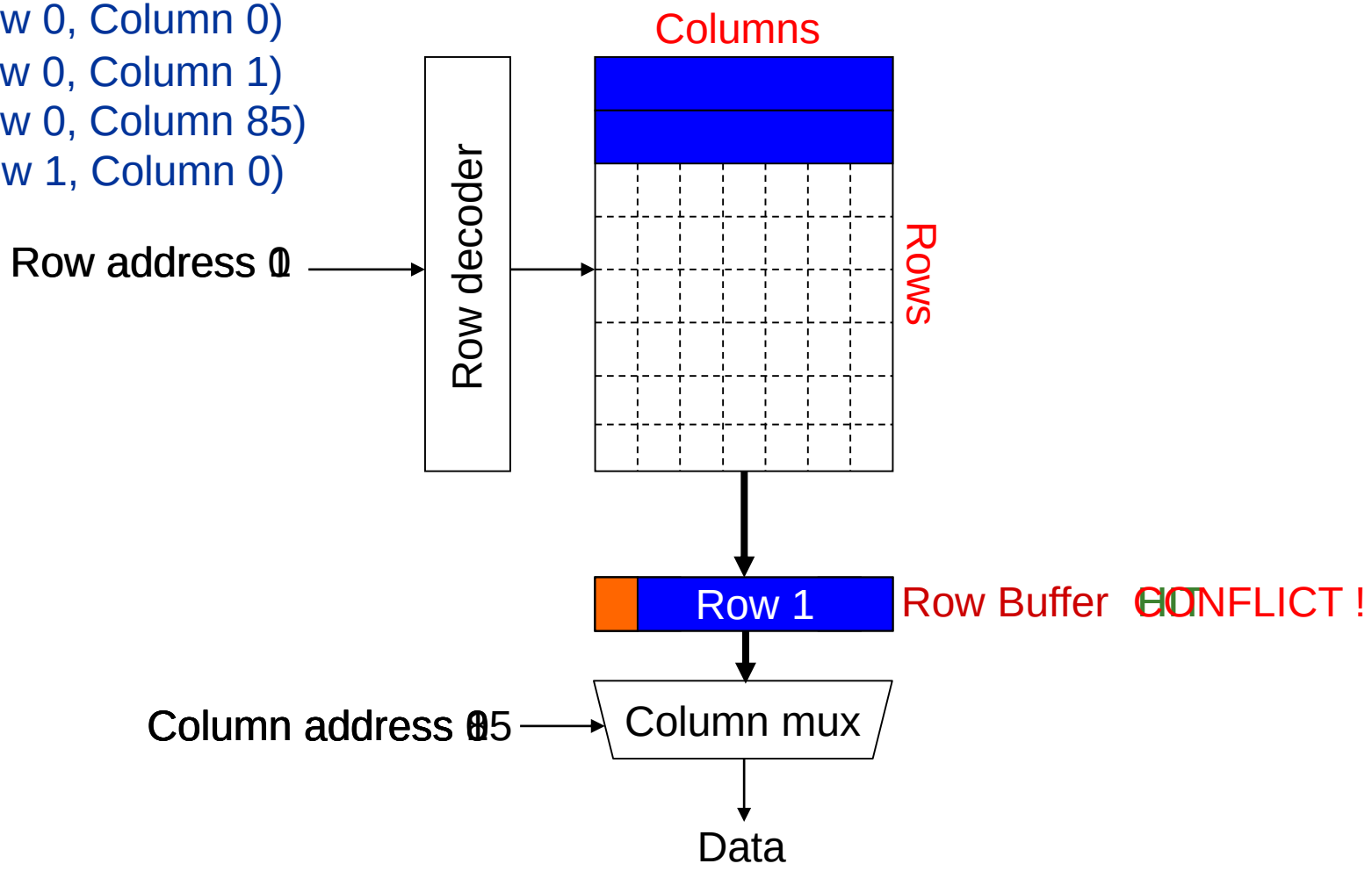


# Page Mode DRAM

- A DRAM bank is a 2D array of cells: rows x columns
- A “DRAM row” is also called a “DRAM page”
- “Sense amplifiers” also called “row buffer”
- Each address is a <row,column> pair
- Access to a “closed row”
  - Activate command opens row (placed into row buffer)
  - Read/write command reads/writes column in the row buffer
  - Precharge command closes the row and prepares the bank for next access
- Access to an “open row”
  - No need for activate command

# DRAM Bank Operation

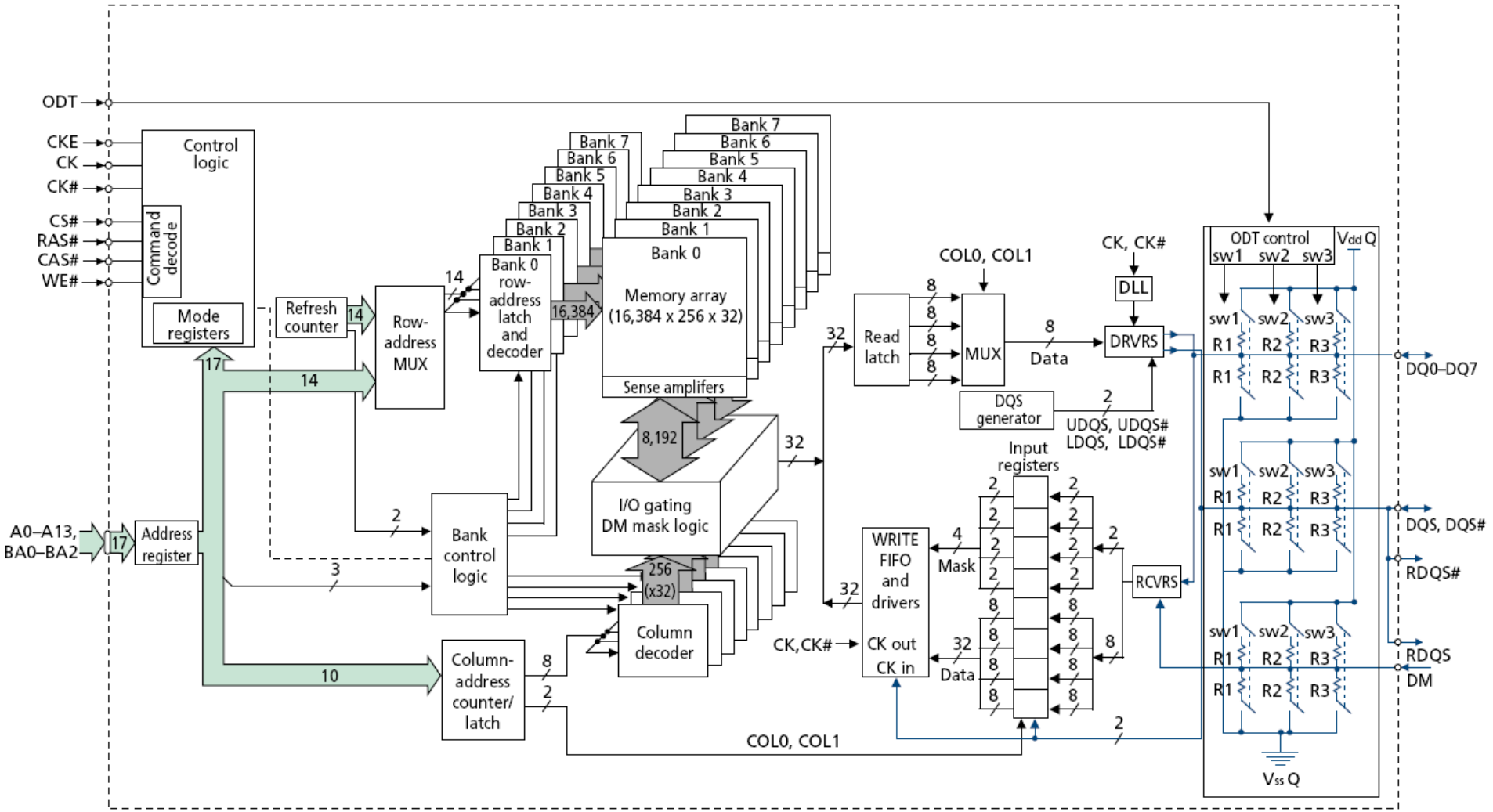
Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)



# The DRAM Chip

- Consists of multiple banks (2-16 in Synchronous DRAM)
- Banks share command/address/data buses
- The chip itself has a narrow interface (4-16 bits per read)

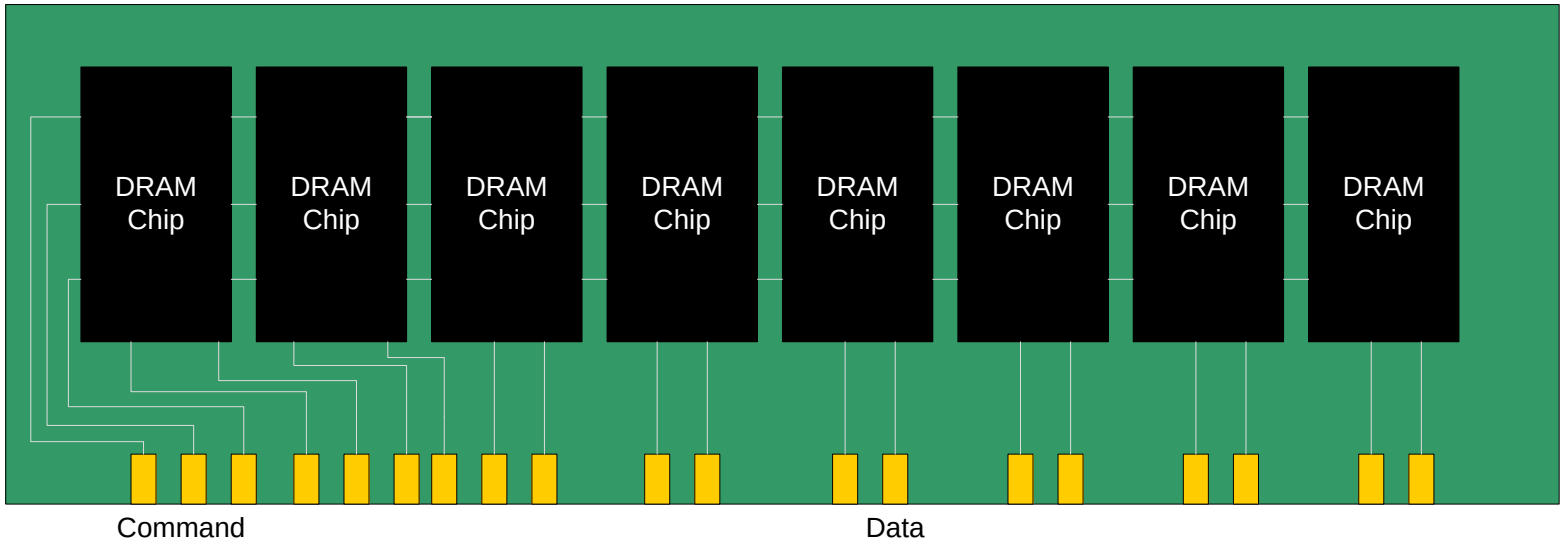
# 128M x 8-bit DRAM Chip



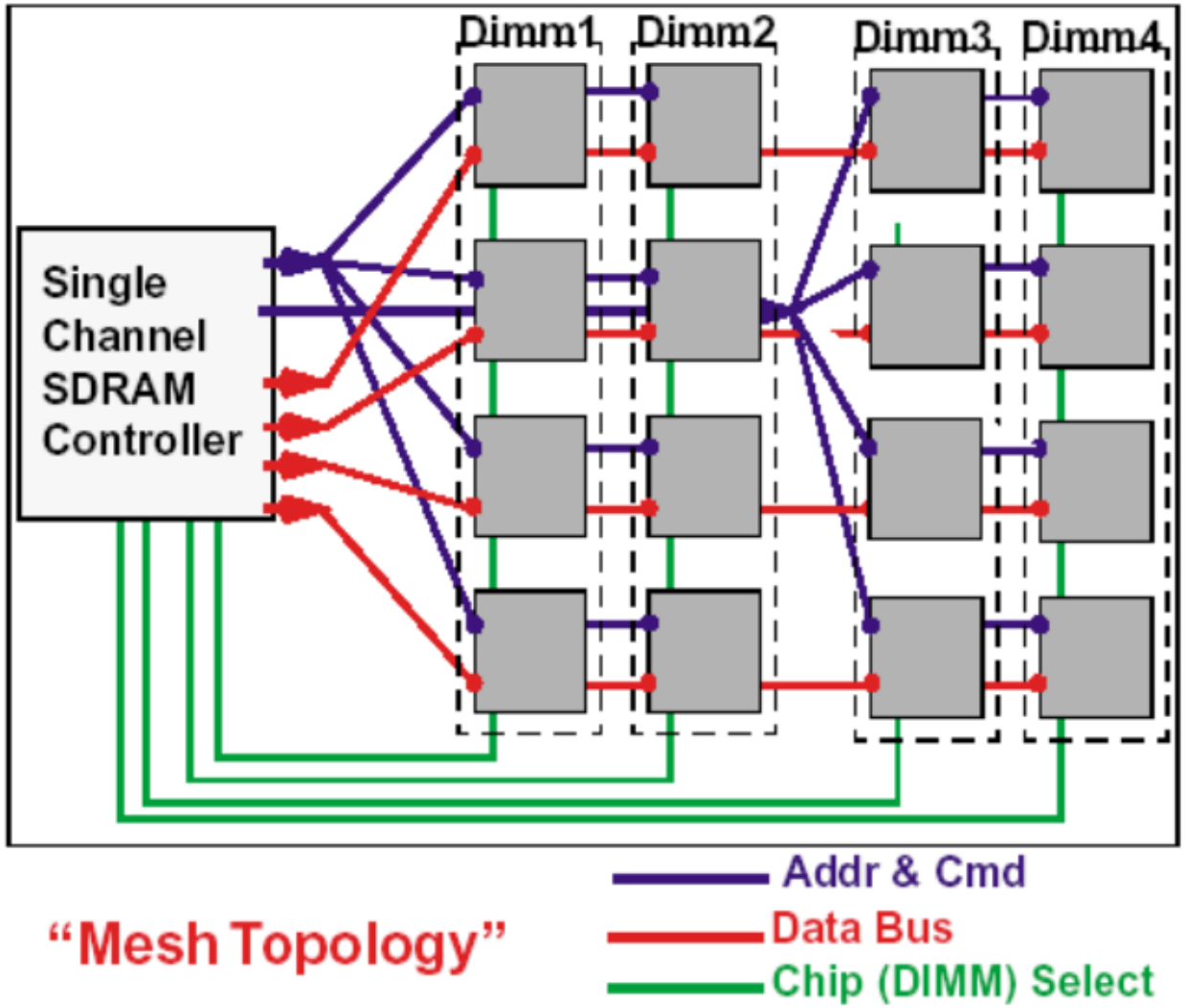
# DRAM Rank and Module

- Rank: Multiple chips operated together to form a wide interface
- All chips comprising a rank are controlled at the same time
  - Respond to a single command
  - Share address and command buses, but provide different data
  - Like DRAM "SIMD"
- A DRAM module consists of one or more ranks
  - E.g., DIMM (dual inline memory module)
  - This is what you plug into your motherboard
- If we have chips with 8-bit interface, to read 8 bytes in a single access, use 8 chips in a DIMM

# A 64-bit Wide DIMM (One Rank)



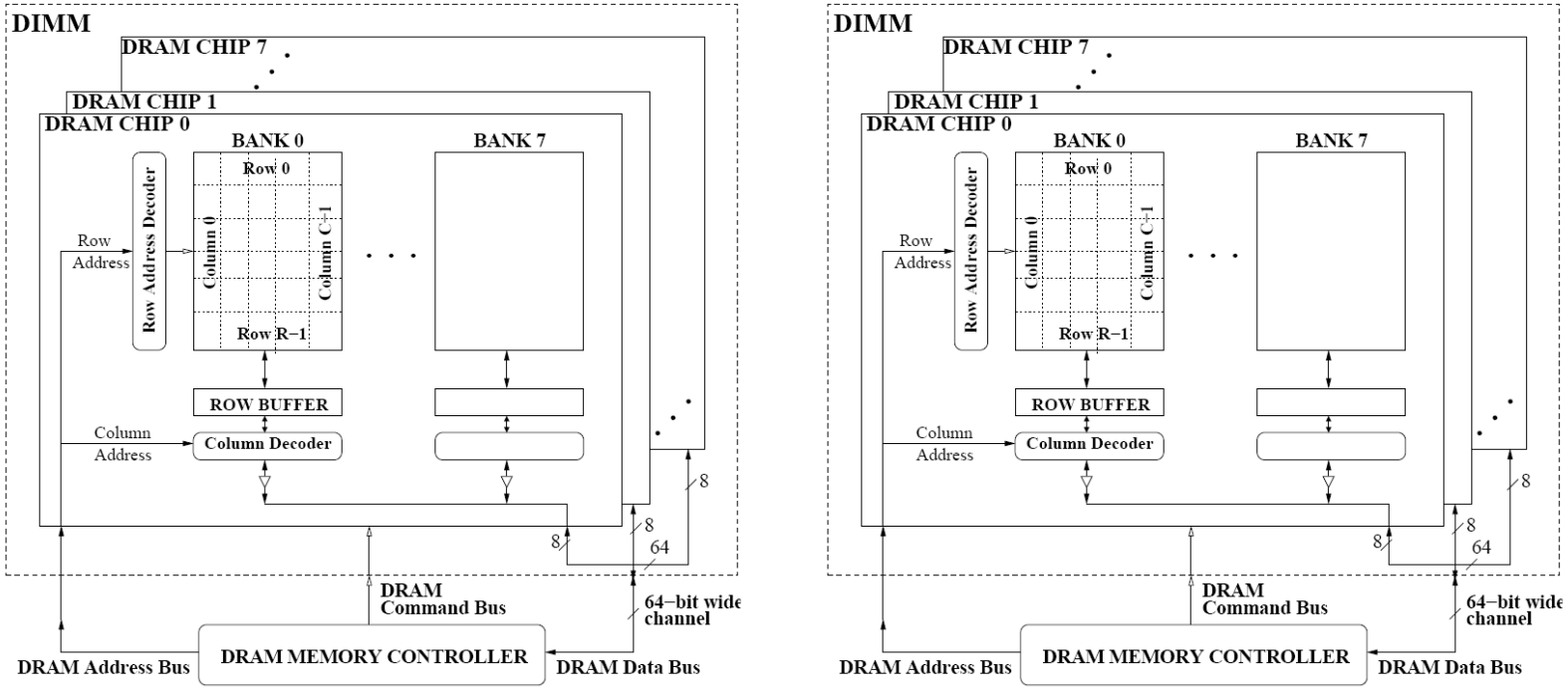
# Multiple DIMMs




"Mesh Topology"

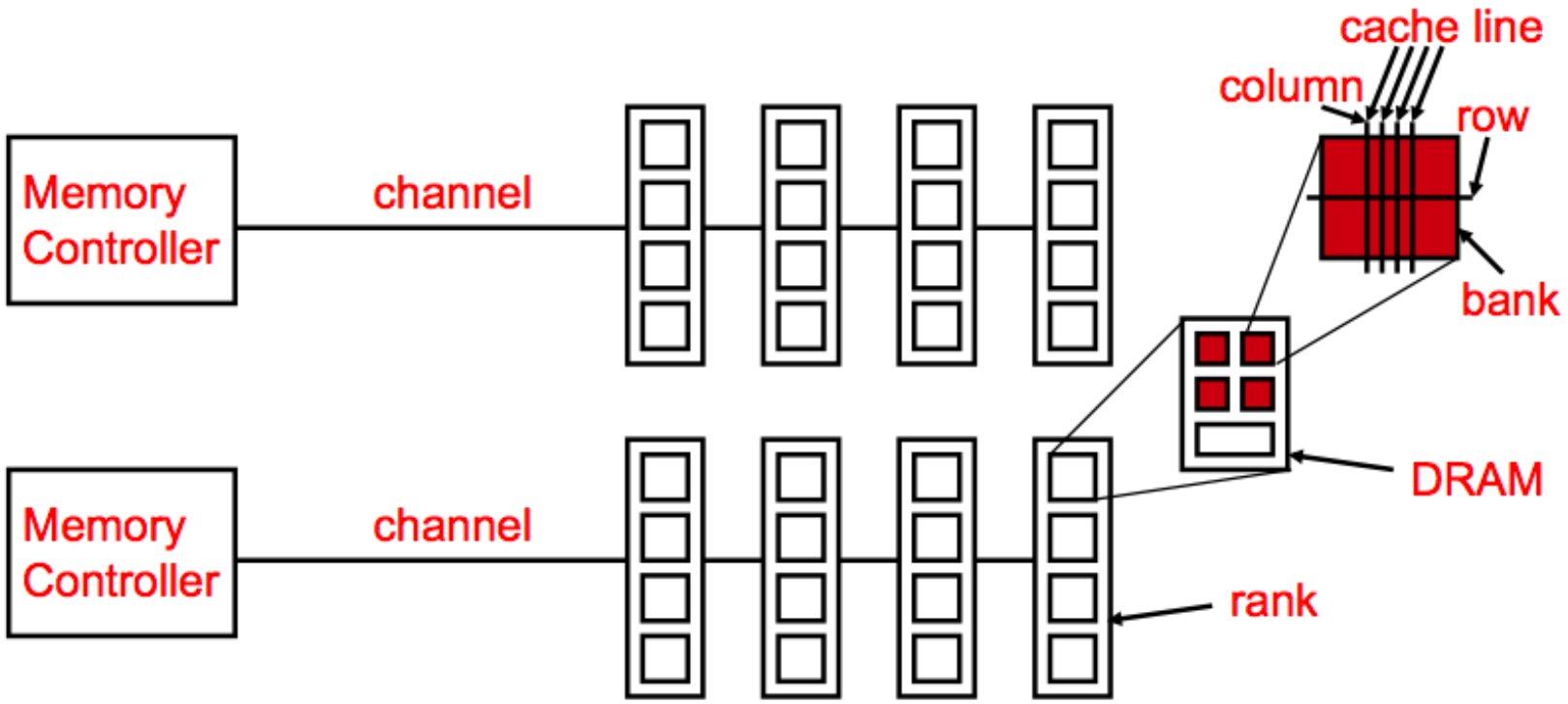
- Advantages:
  - Enables even higher capacity
- Disadvantages:
  - Interconnect complexity and energy consumption can be high

# DRAM Channels

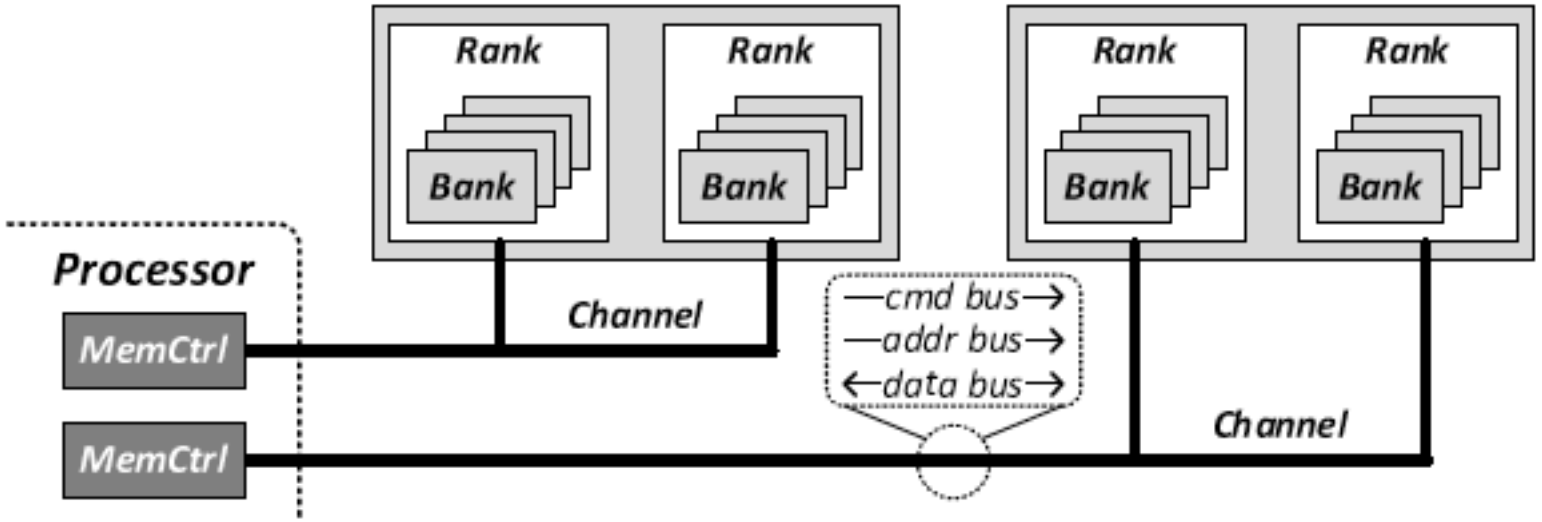



- 2 Independent Channels: 2 Memory Controllers (Above)
- 2 Dependent/Lockstep Channels: 1 Memory Controller with wide interface (Not Shown above)

# Generalized Memory Structure



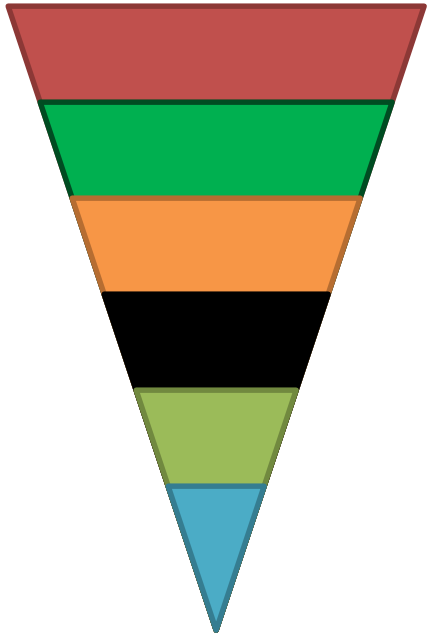
# Generalized Memory Structure



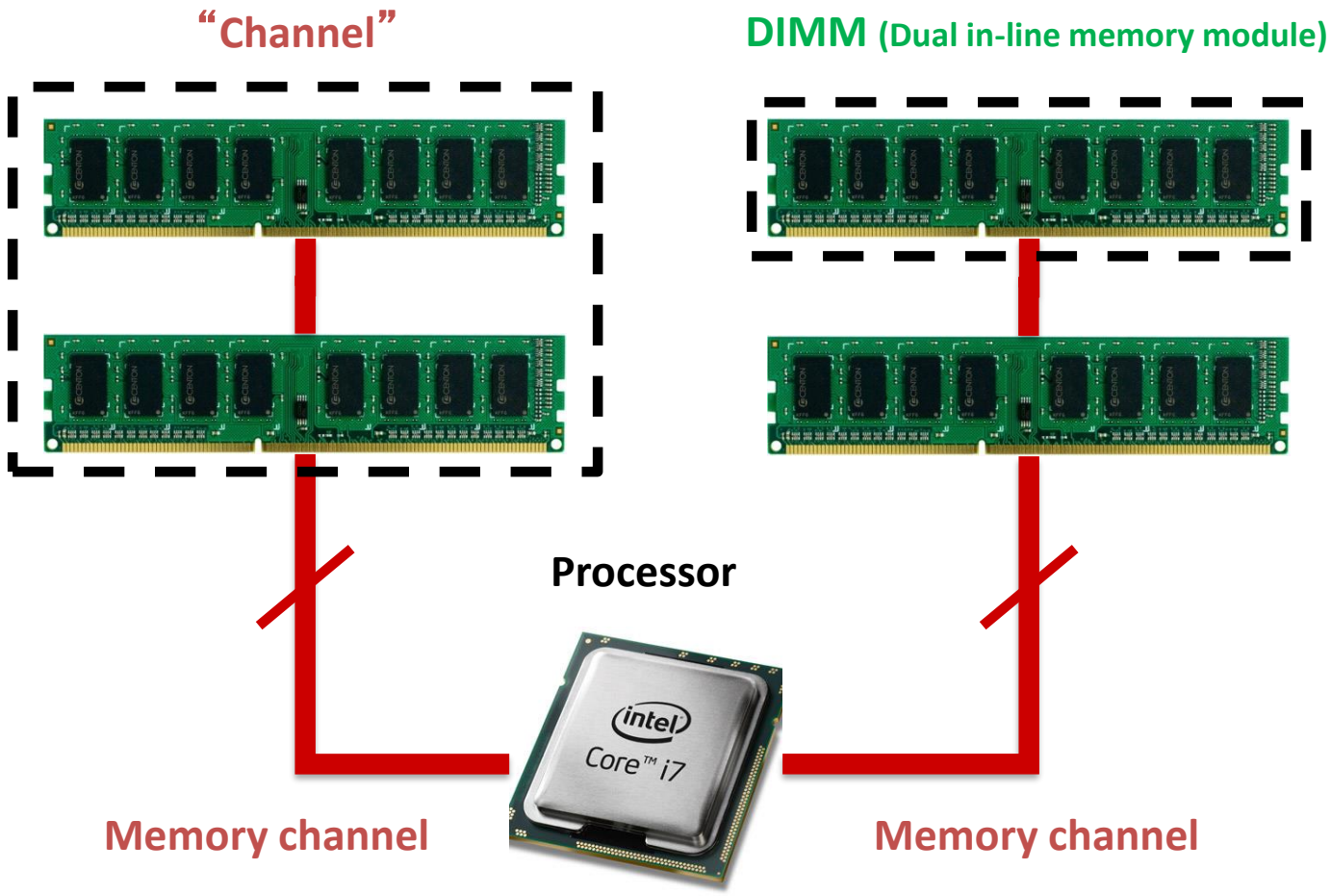
# The DRAM Subsystem
# The Top Down View

# DRAM Subsystem Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column



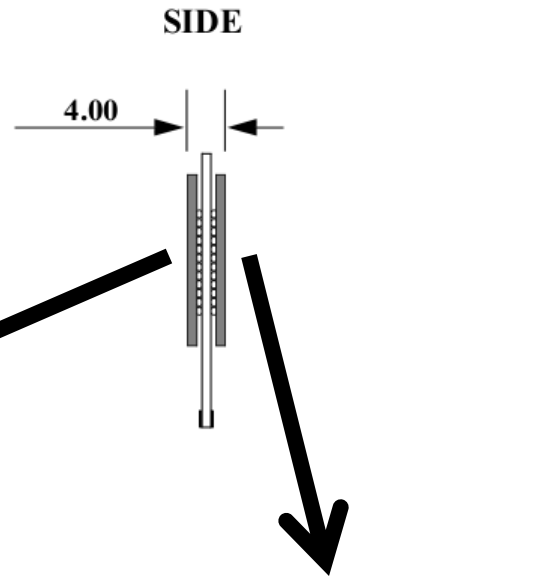
# The DRAM subsystem



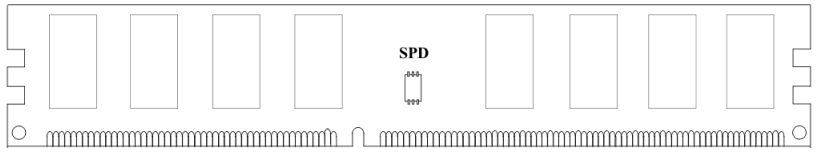
# Breaking down a DIMM

**DIMM** (Dual in-line memory module)

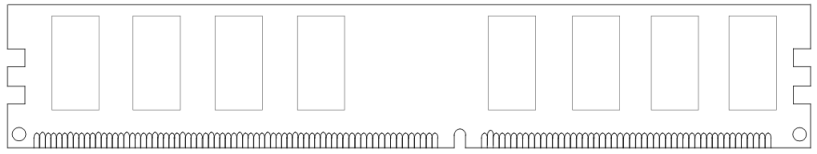
Side view




**Front of DIMM**



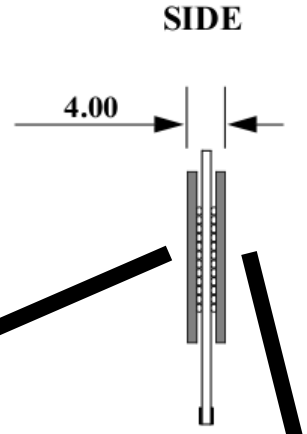

**Back of DIMM**



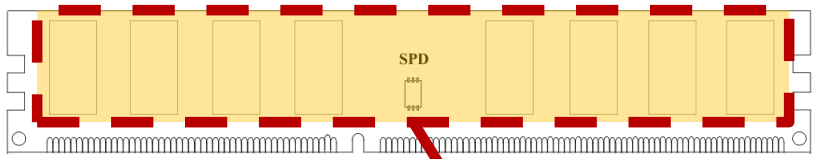
# Breaking down a DIMM

**DIMM** (Dual in-line memory module)

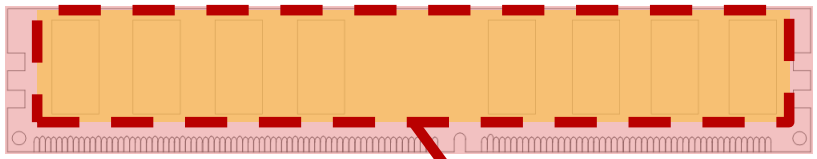
Side view




**Front of DIMM**



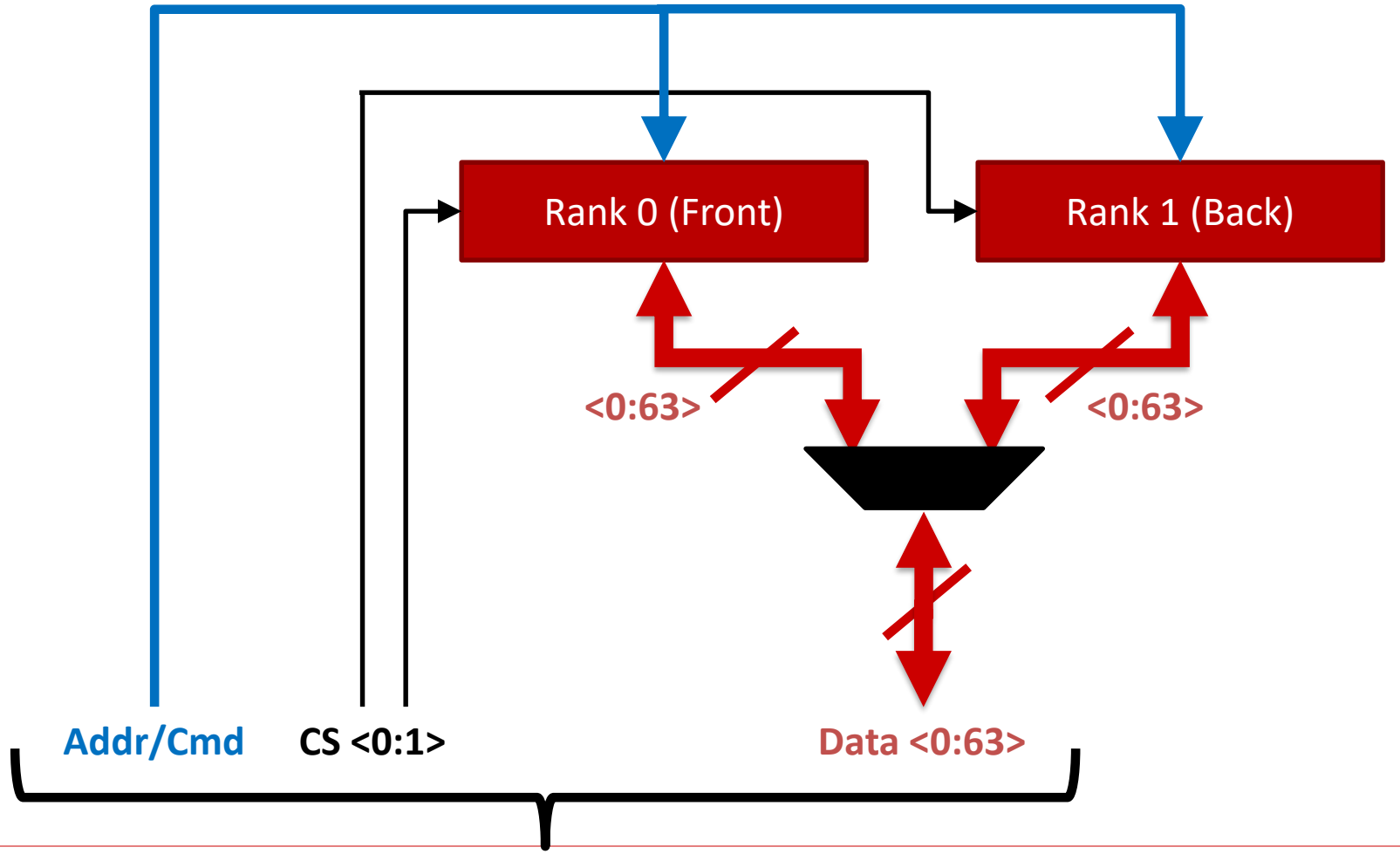

**Back of DIMM**



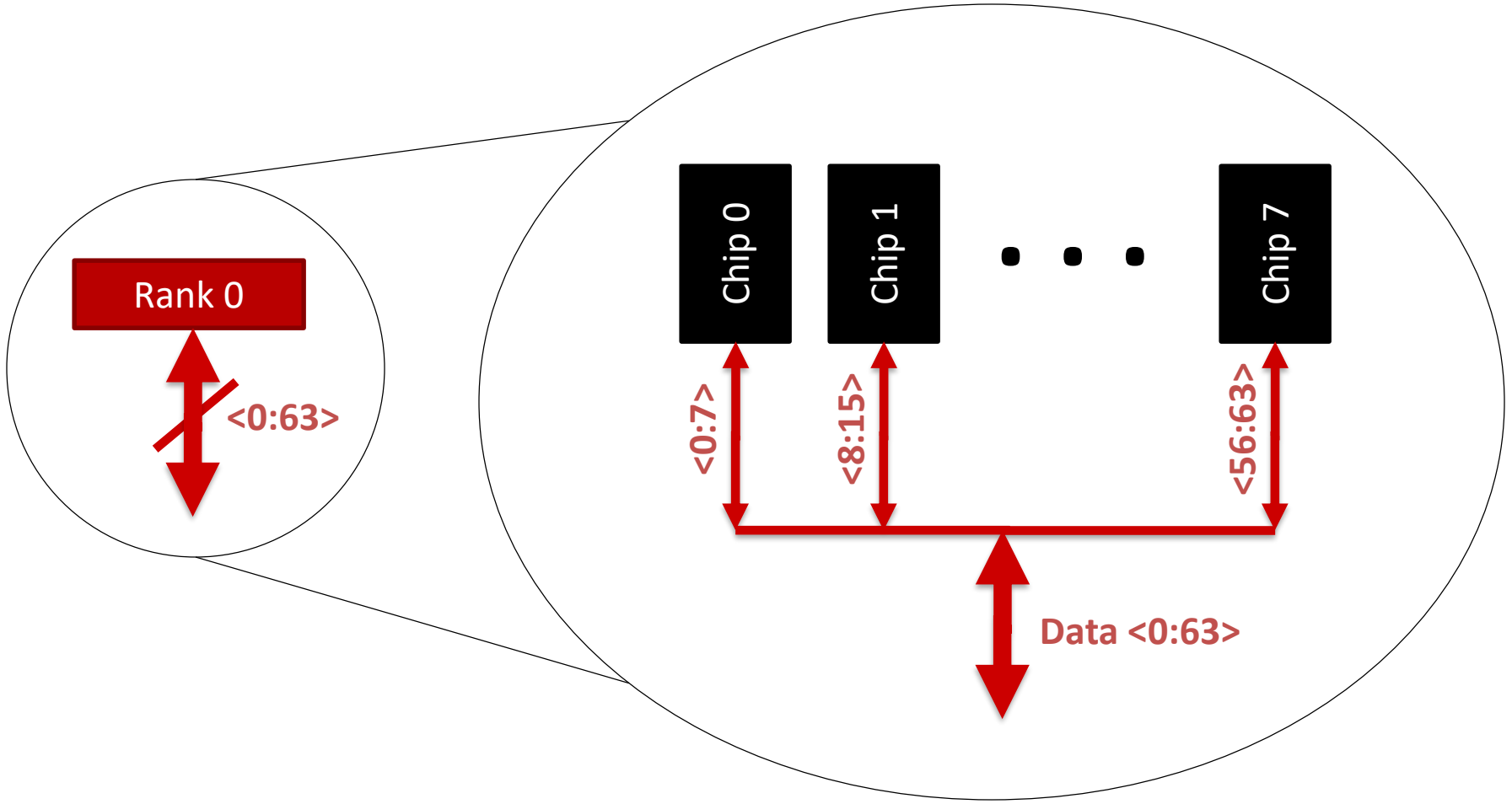
**Rank 0:** collection of 8 chips

**Rank 1**

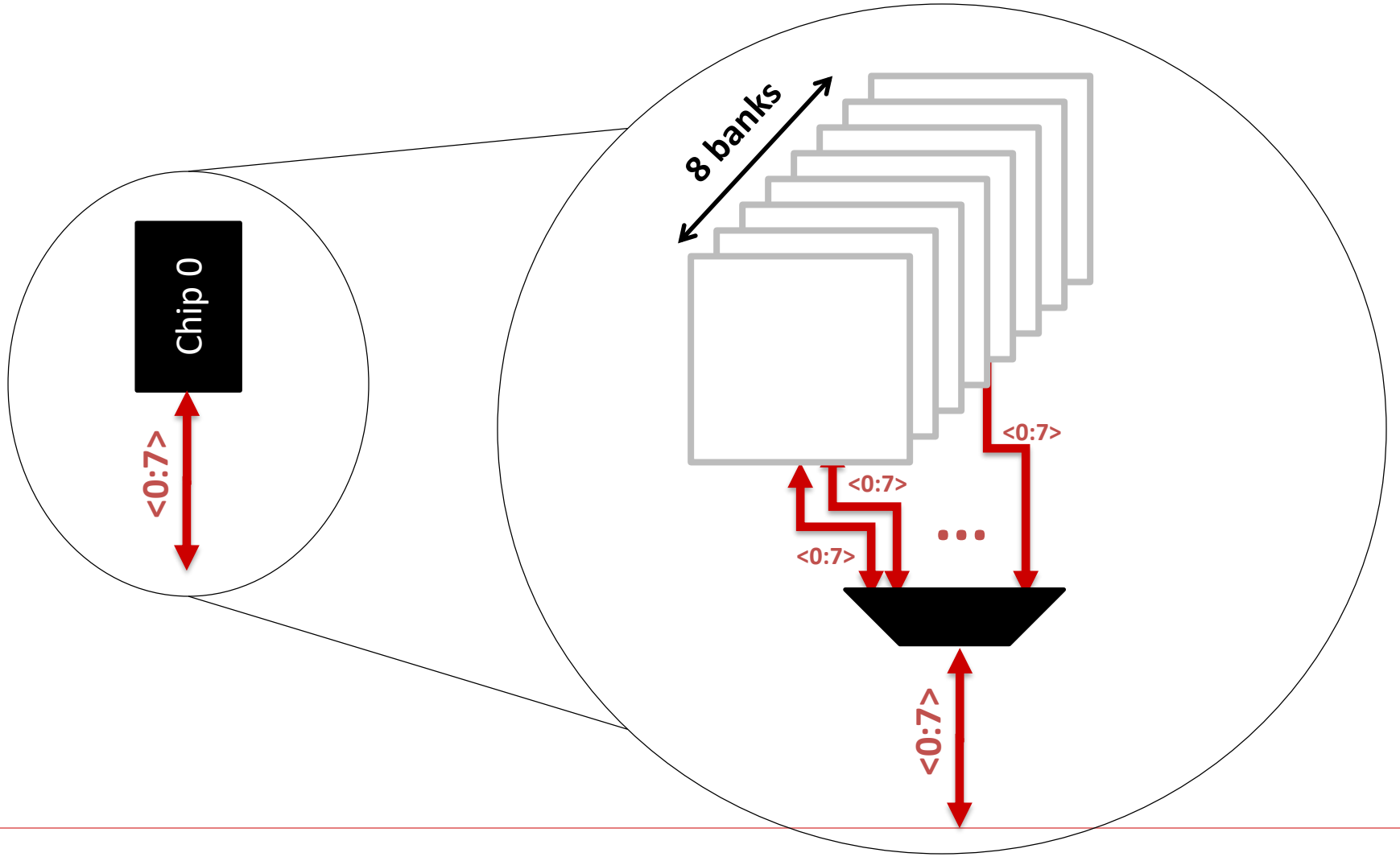
# Rank


Rank 0 (Front)
Rank 1 (Back)
<0:63>
<0:63>
Addr/Cmd
CS <0:1>
Data <0:63>
Memory channel

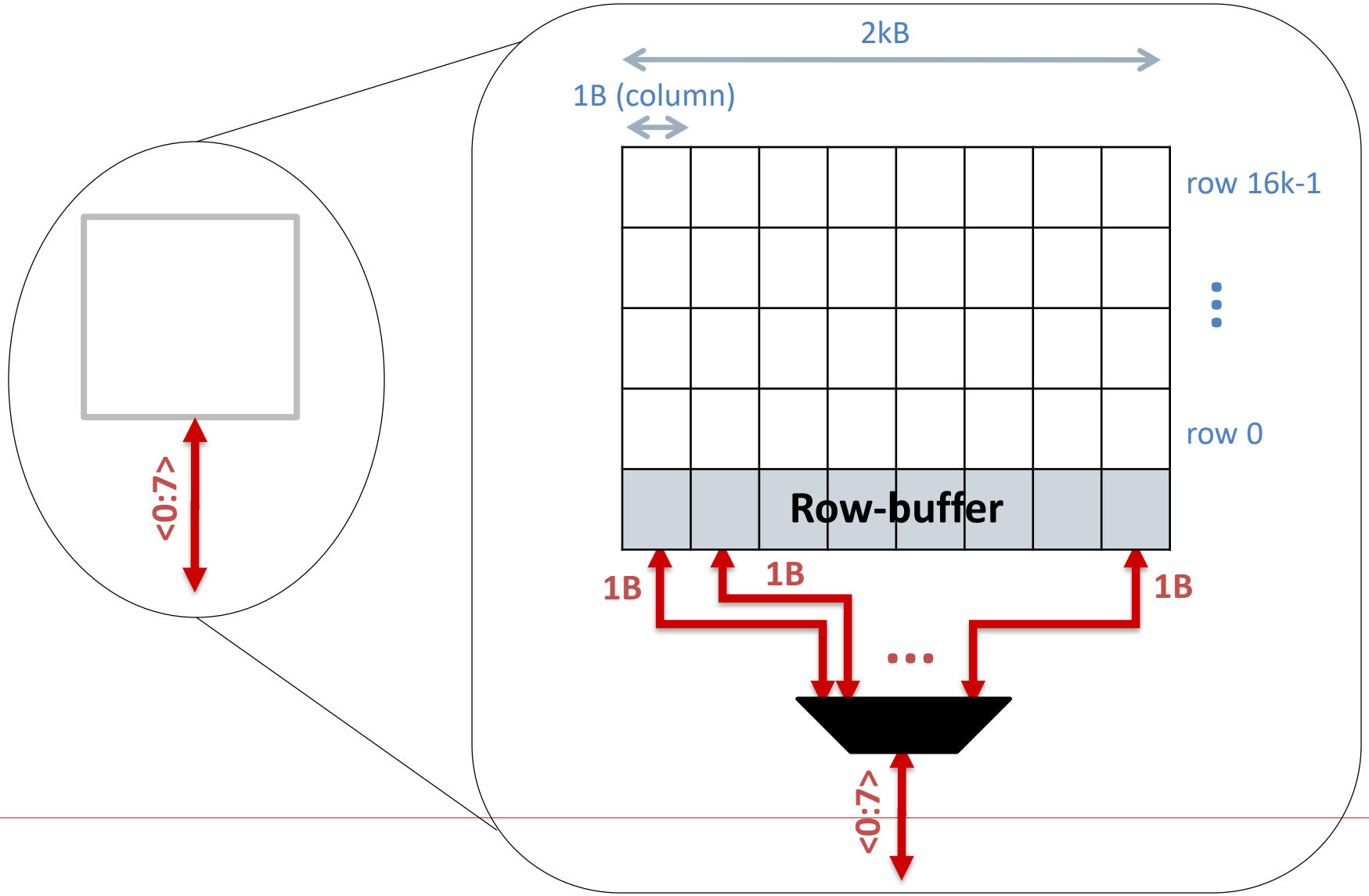
# Breaking down a Rank



# Breaking down a Chip

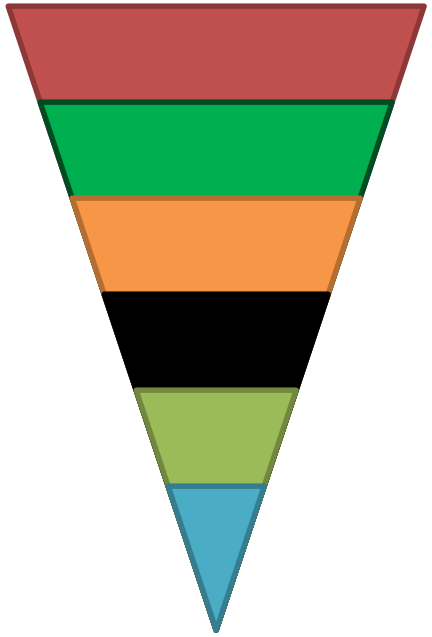


# Breaking down a Bank

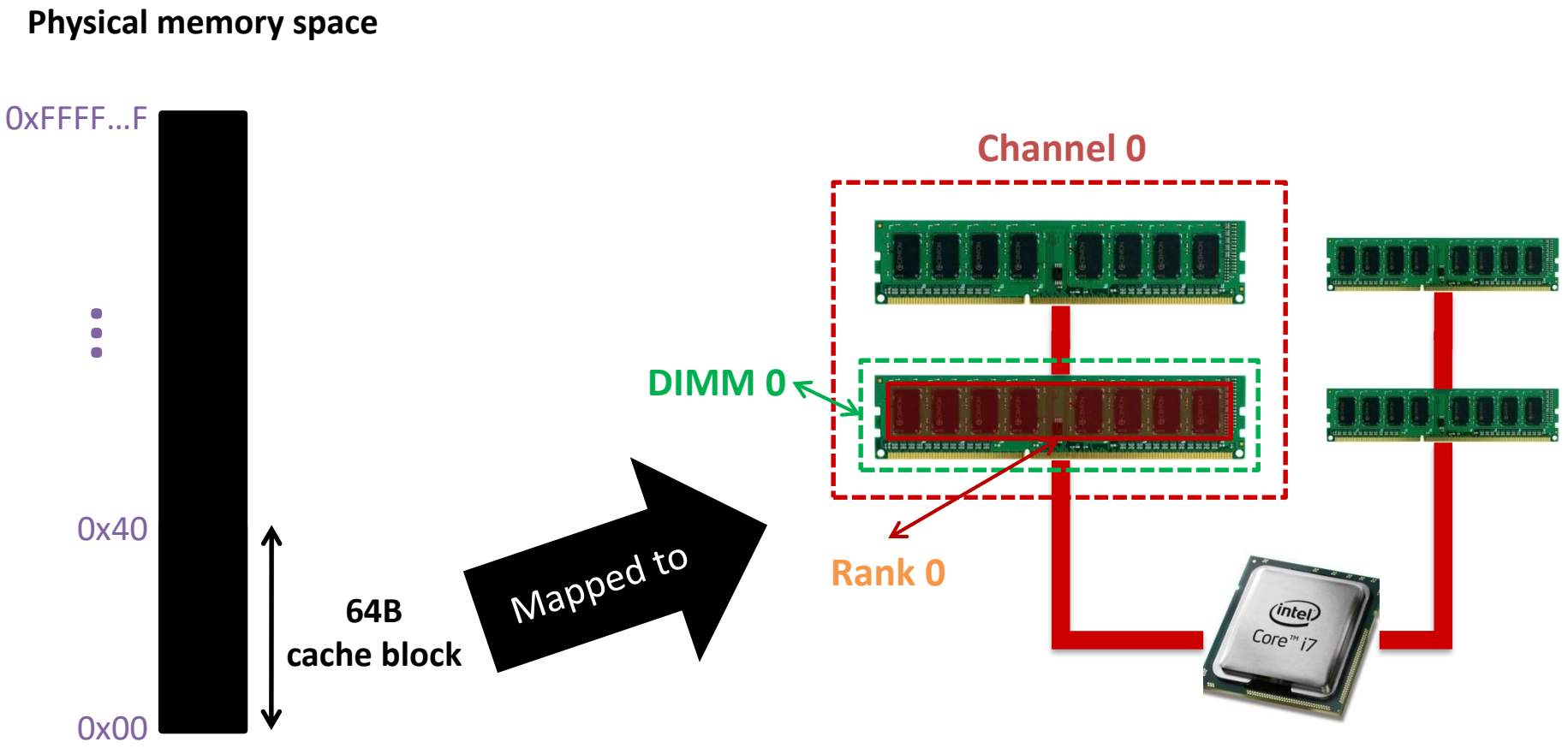


# DRAM Subsystem Organization

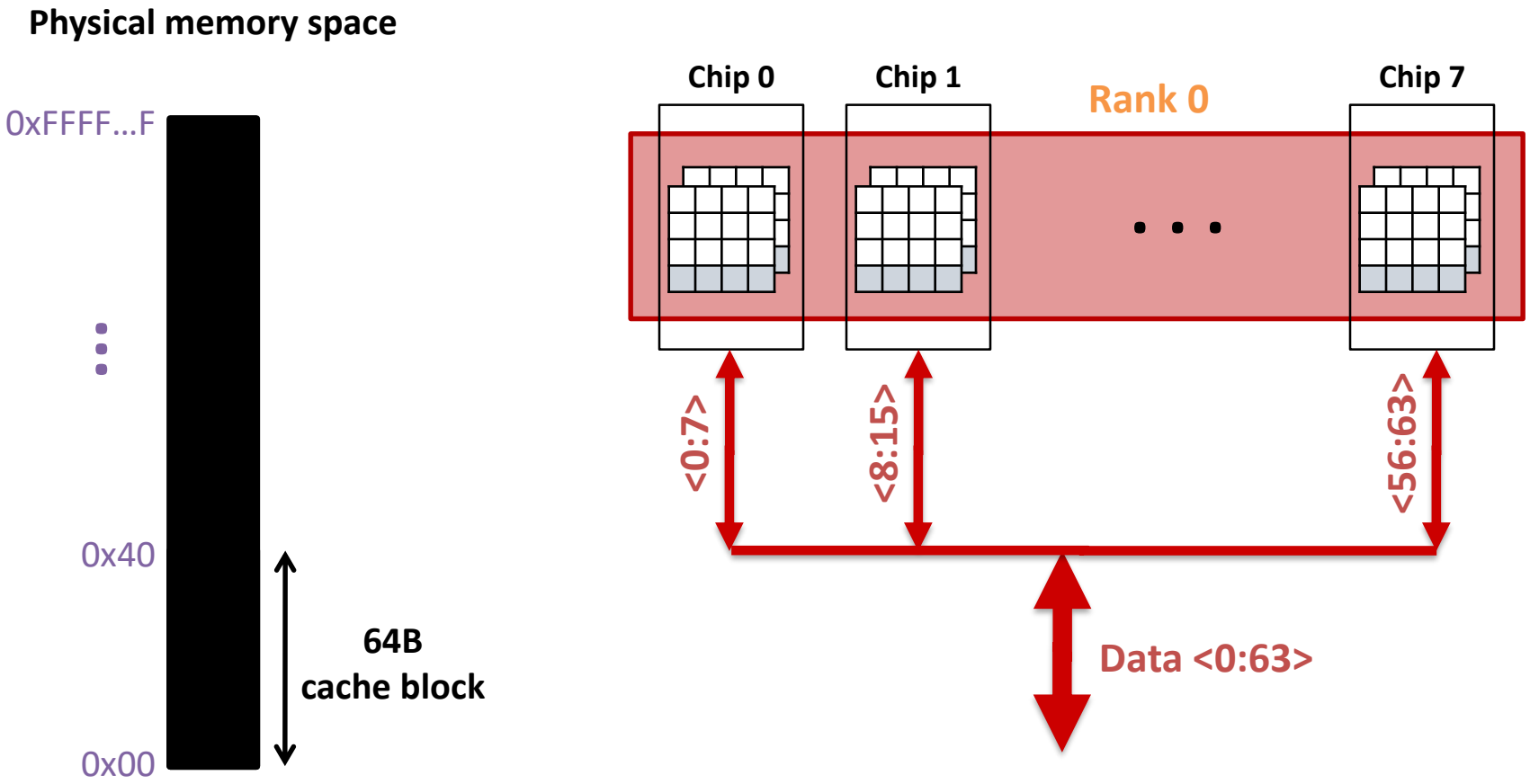
- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column



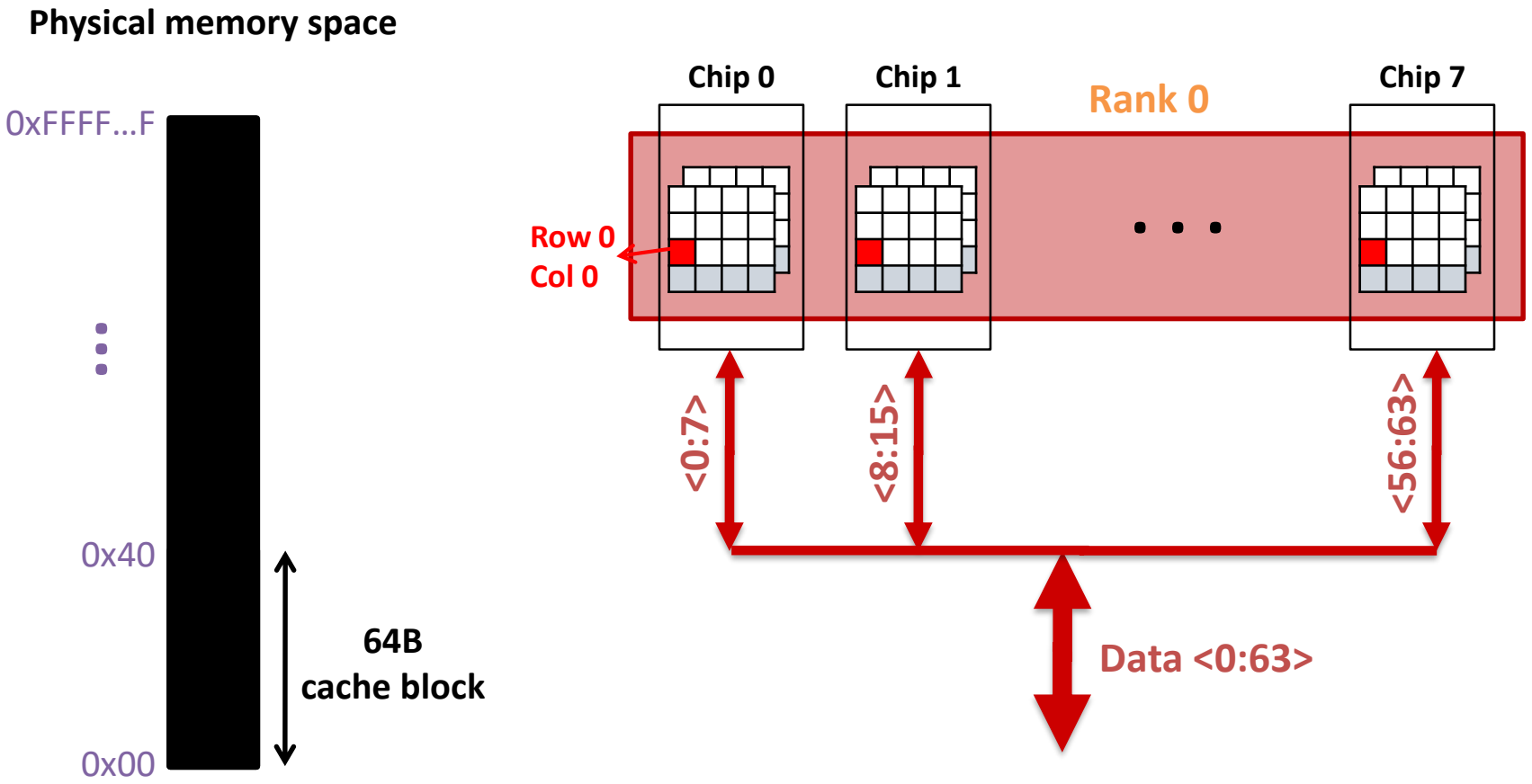
# Example: Transferring a cache block



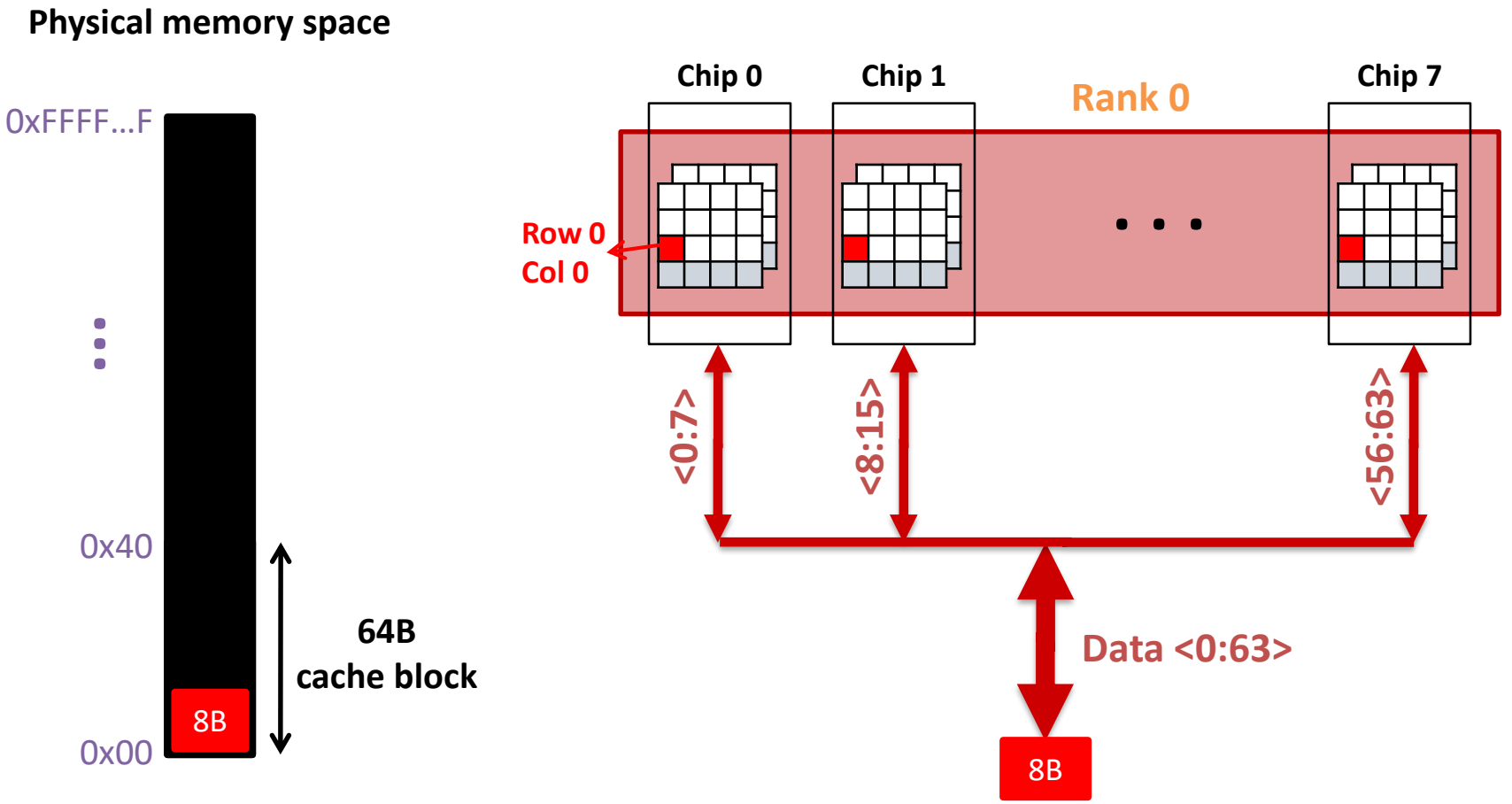
# Example: Transferring a cache block



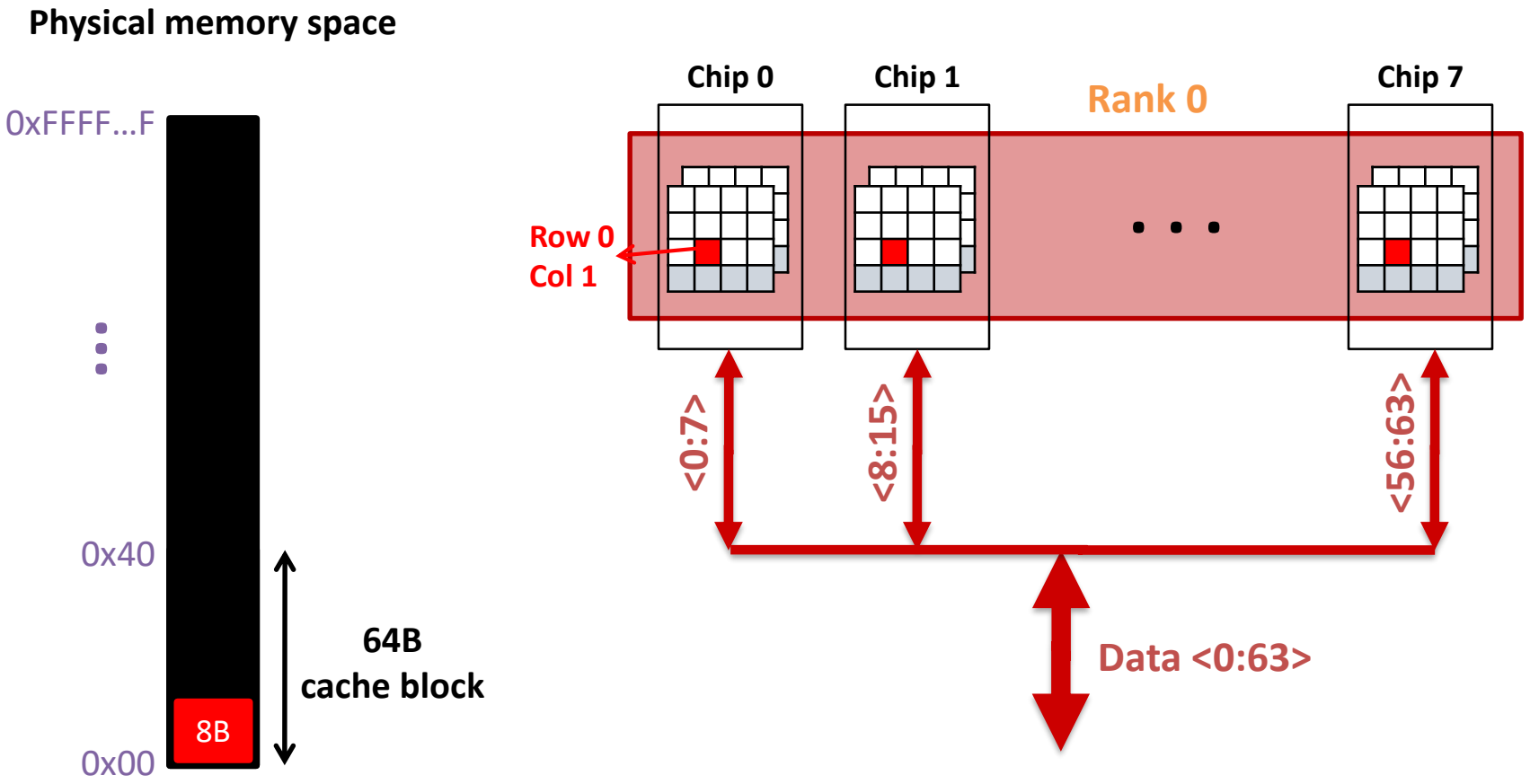
# Example: Transferring a cache block



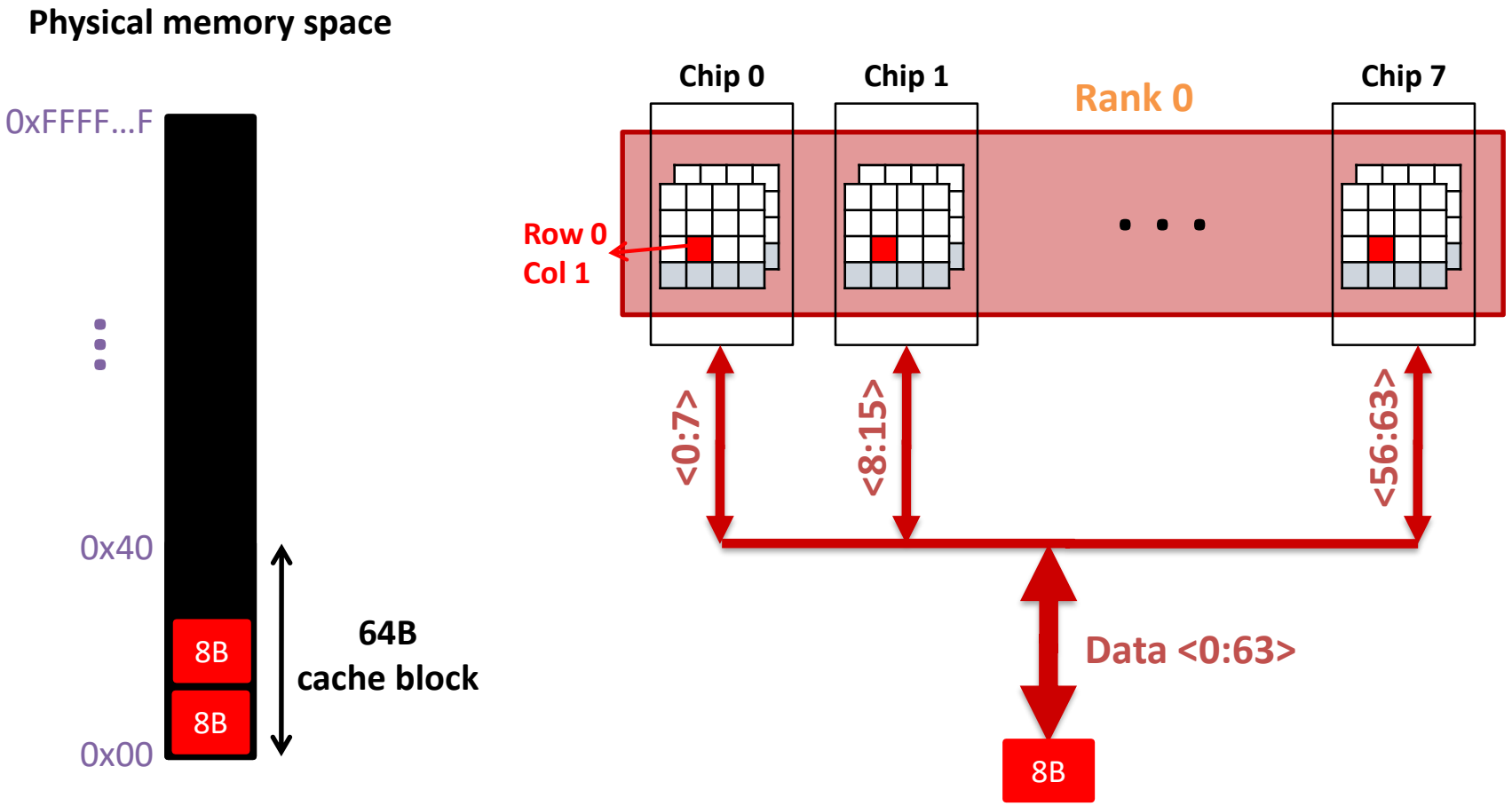
# Example: Transferring a cache block



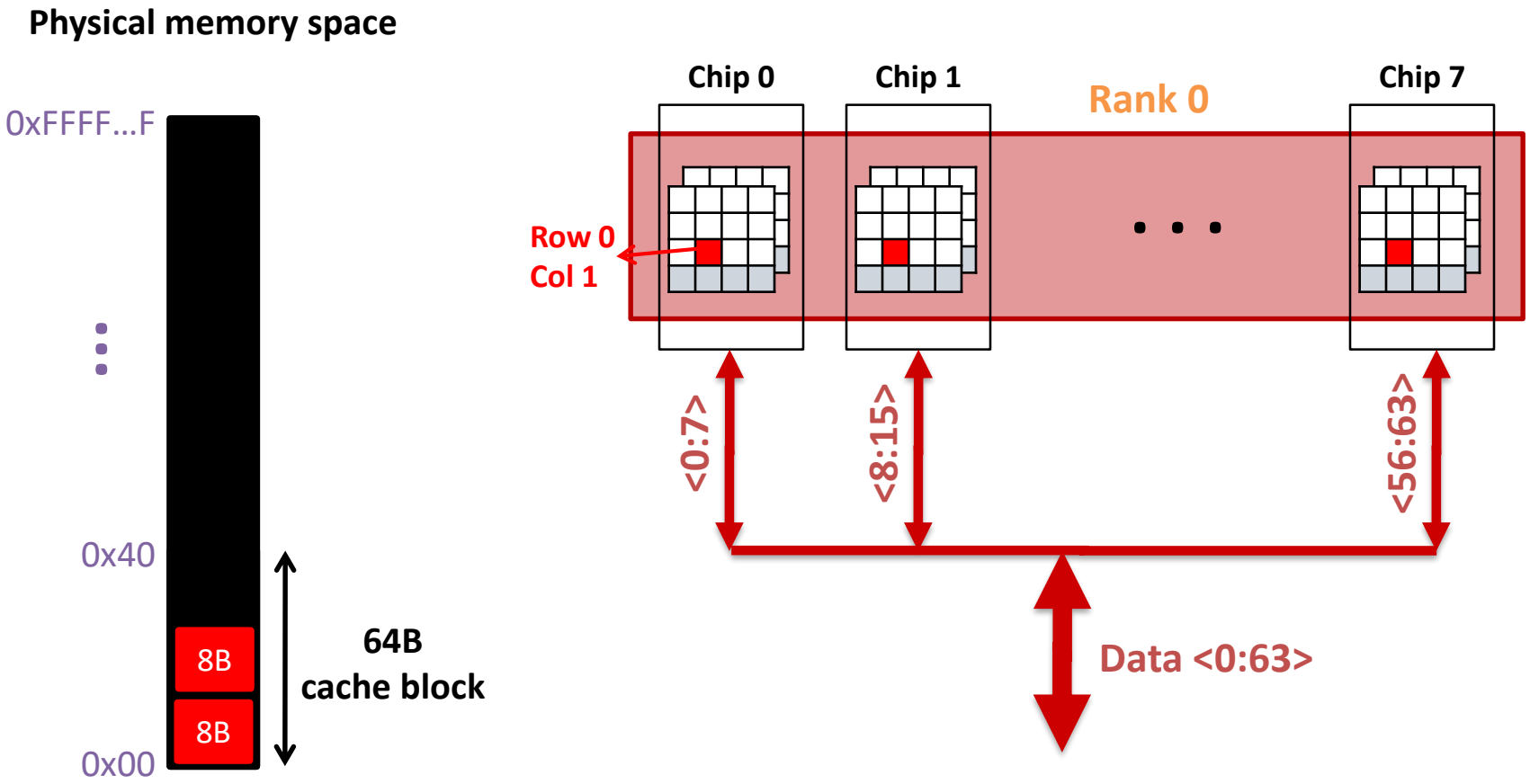
# Example: Transferring a cache block



# Example: Transferring a cache block



# Example: Transferring a cache block




**A 64B cache block takes 8 I/O cycles to transfer.**

**During the process, 8 columns are read sequentially.**

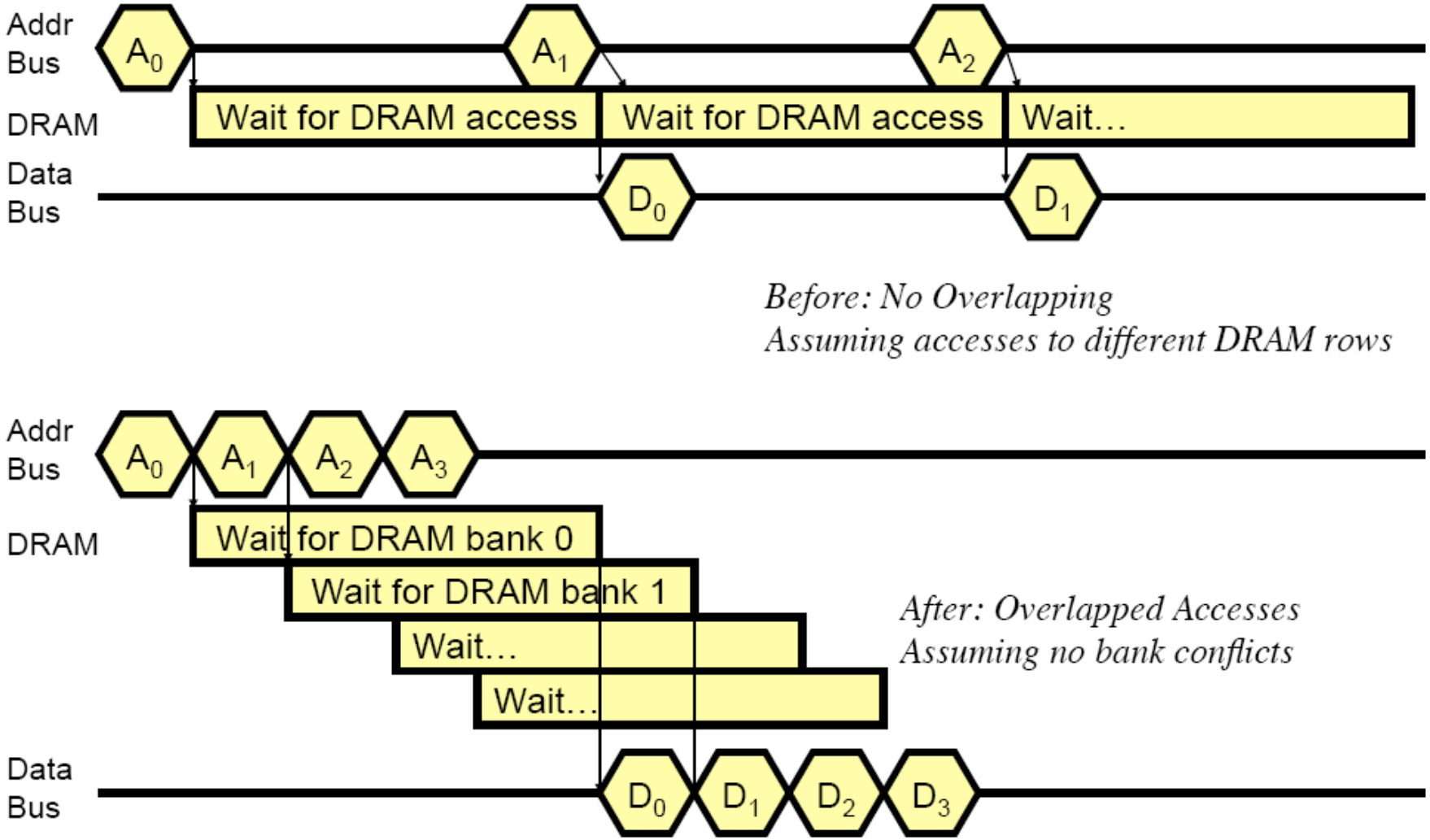
# Latency Components: Basic DRAM Operation

- CPU → controller transfer time
- Controller latency
  - Queuing & scheduling delay at the controller
  - Access converted to basic commands
- Controller → DRAM transfer time
- DRAM bank latency
  - Simple CAS if row is "open" OR
  - RAS + CAS if array precharged OR
  - PRE + RAS + CAS (worst case)
- DRAM → CPU transfer time (through controller)

# Multiple Banks (Interleaving) and Channels

- Multiple banks
  - Enable concurrent DRAM accesses
  - Bits in address determine which bank an address resides in
- Multiple independent channels serve the same purpose
  - But they are even better because they have separate data buses
  - Increased bus bandwidth
- Enabling more concurrency requires reducing
  - Bank conflicts
  - Channel conflicts
- How to select/randomize bank/channel indices in address?
  - Lower order bits have more entropy
  - Randomizing hash functions (XOR of different address bits)

# How Multiple Banks/Channels Help



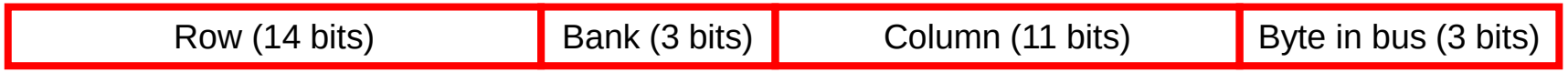
# Multiple Channels

- Advantages
  - Increased bandwidth
  - Multiple concurrent accesses (if independent channels)

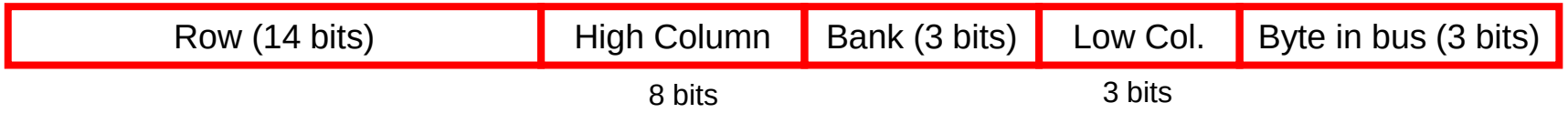
- Disadvantages
  - Higher cost than a single channel
    - More board wires
    - More pins (if on-chip memory controller)

# Address Mapping (Single Channel)

- Single-channel system with 8-byte memory bus
  - 2GB memory, 8 banks, 16K rows & 2K columns per bank
- Row interleaving
  - Consecutive rows of memory in consecutive banks



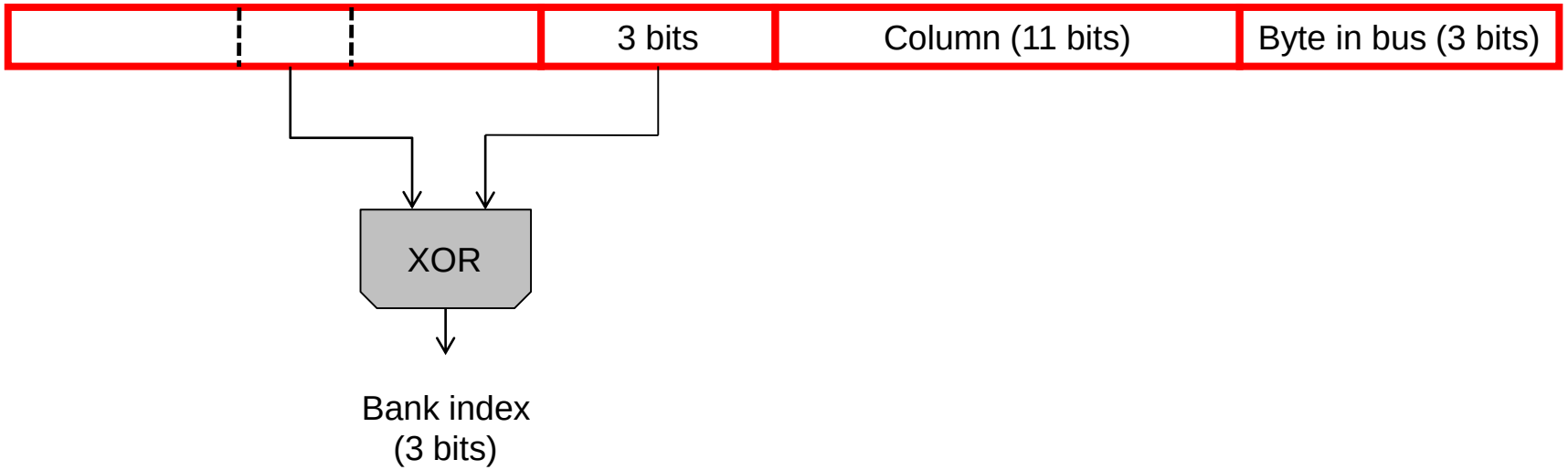

- Cache block interleaving
  - Consecutive cache block addresses in consecutive banks
  - 64 byte cache blocks



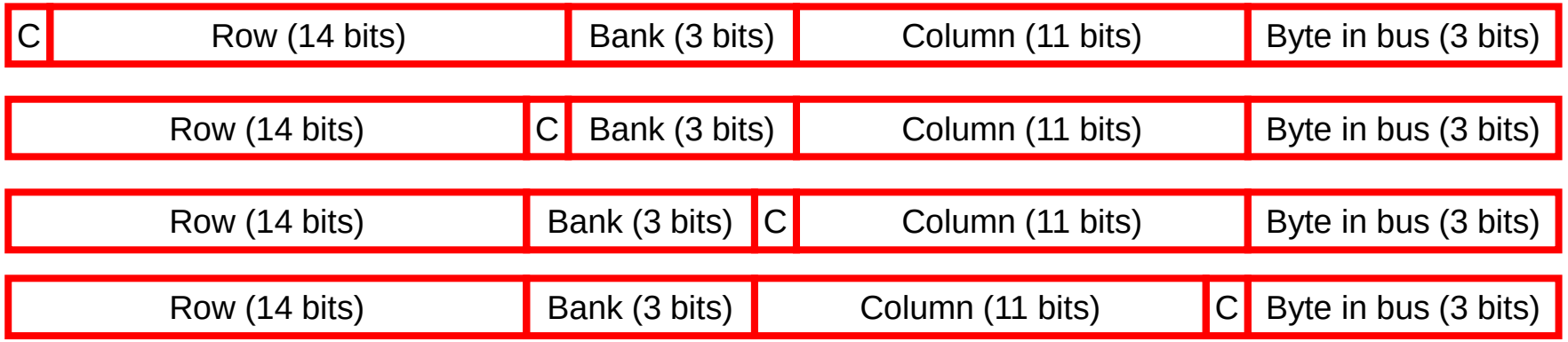

  - Accesses to consecutive cache blocks can be serviced in parallel
  - How about random accesses? Strided accesses?

# Bank Mapping Randomization

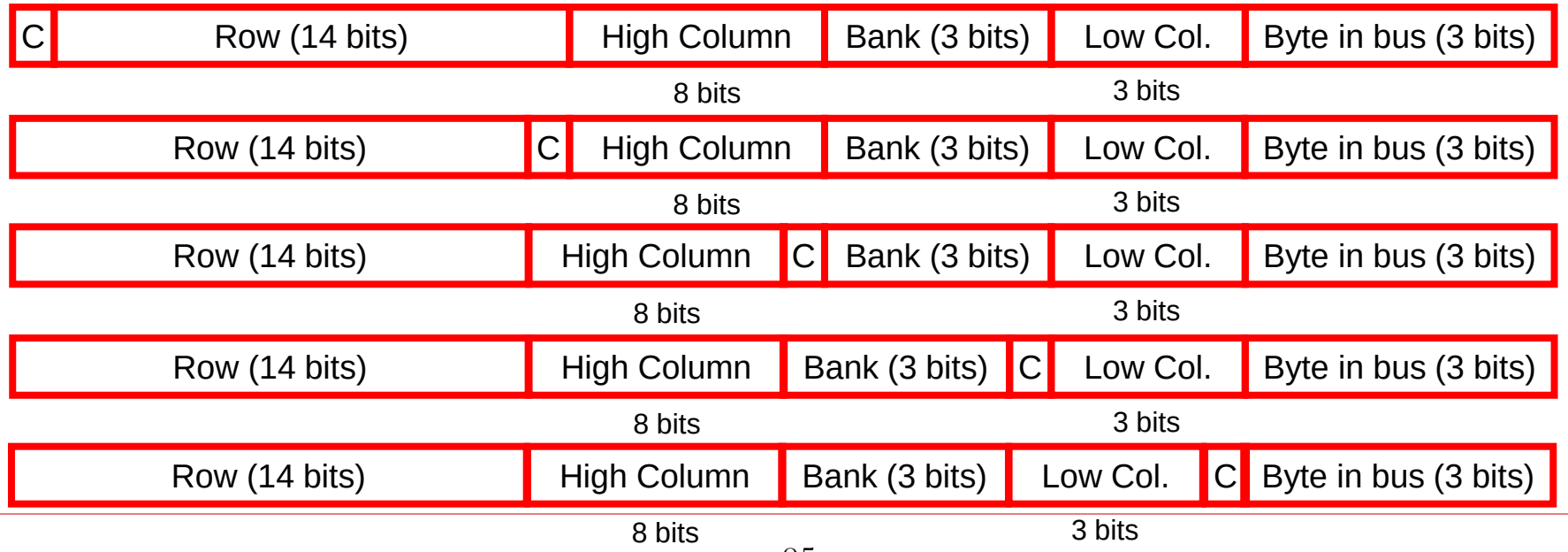
- DRAM controller can randomize the address mapping to banks so that bank conflicts are less likely



# Address Mapping (Multiple Channels)

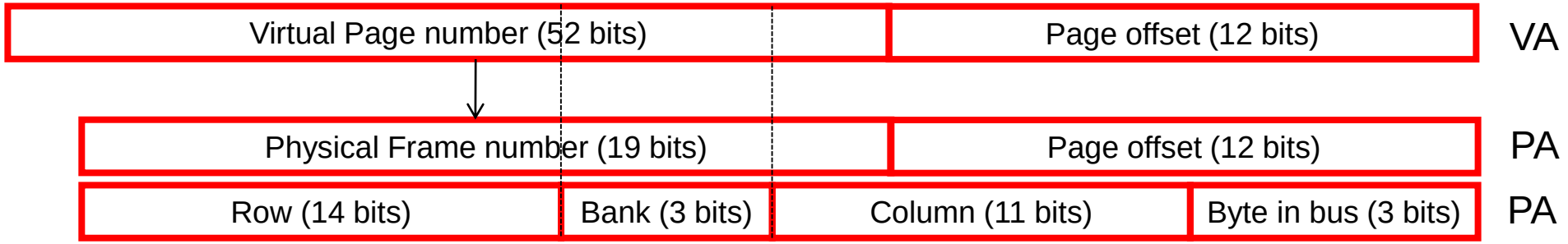



- Where are consecutive cache blocks?



# Interaction with Virtual→Physical Mapping

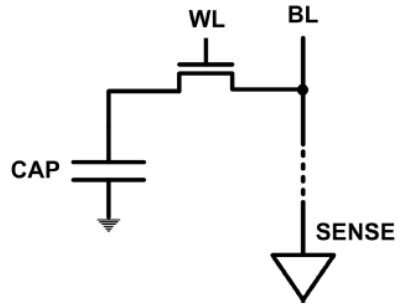
- Operating System influences where an address maps to in DRAM




- Operating system can control which bank/channel/rank a virtual page is mapped to.
- It can perform page coloring to minimize bank conflicts
- Or to minimize inter-application interference

# DRAM Refresh (I)

- DRAM capacitor charge leaks over time
- The memory controller needs to read each row periodically to restore the charge
  - Activate + precharge each row every N ms
  - Typical N = 64 ms



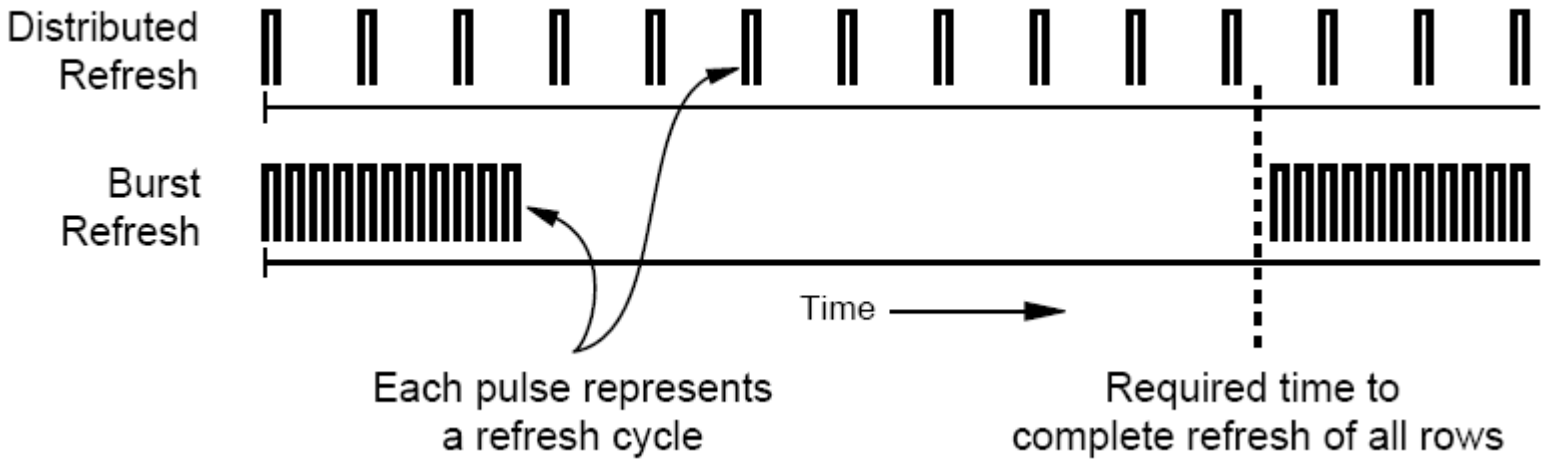

- Implications on performance?

  -- DRAM bank unavailable while refreshed

  -- Long pause times: If we refresh all rows in burst, every 64ms the DRAM will be unavailable until refresh ends

- Burst refresh: All rows refreshed immediately after one another
- Distributed refresh: Each row refreshed at a different time, at regular intervals

# DRAM Refresh (II)




- Distributed refresh eliminates long pause times

# Downsides of DRAM Refresh

- Downsides of refresh

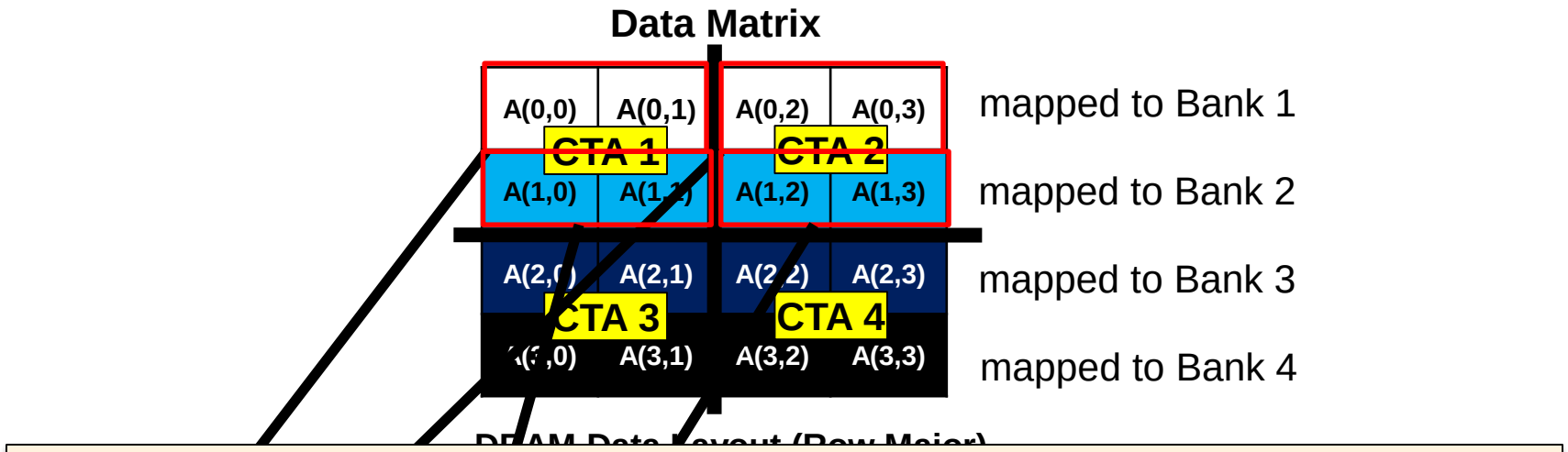
-- Energy consumption: Each refresh consumes energy

-- Performance degradation: DRAM rank/bank unavailable while refreshed

-- QoS/predictability impact: (Long) pause times during refresh

# Back to the paper…

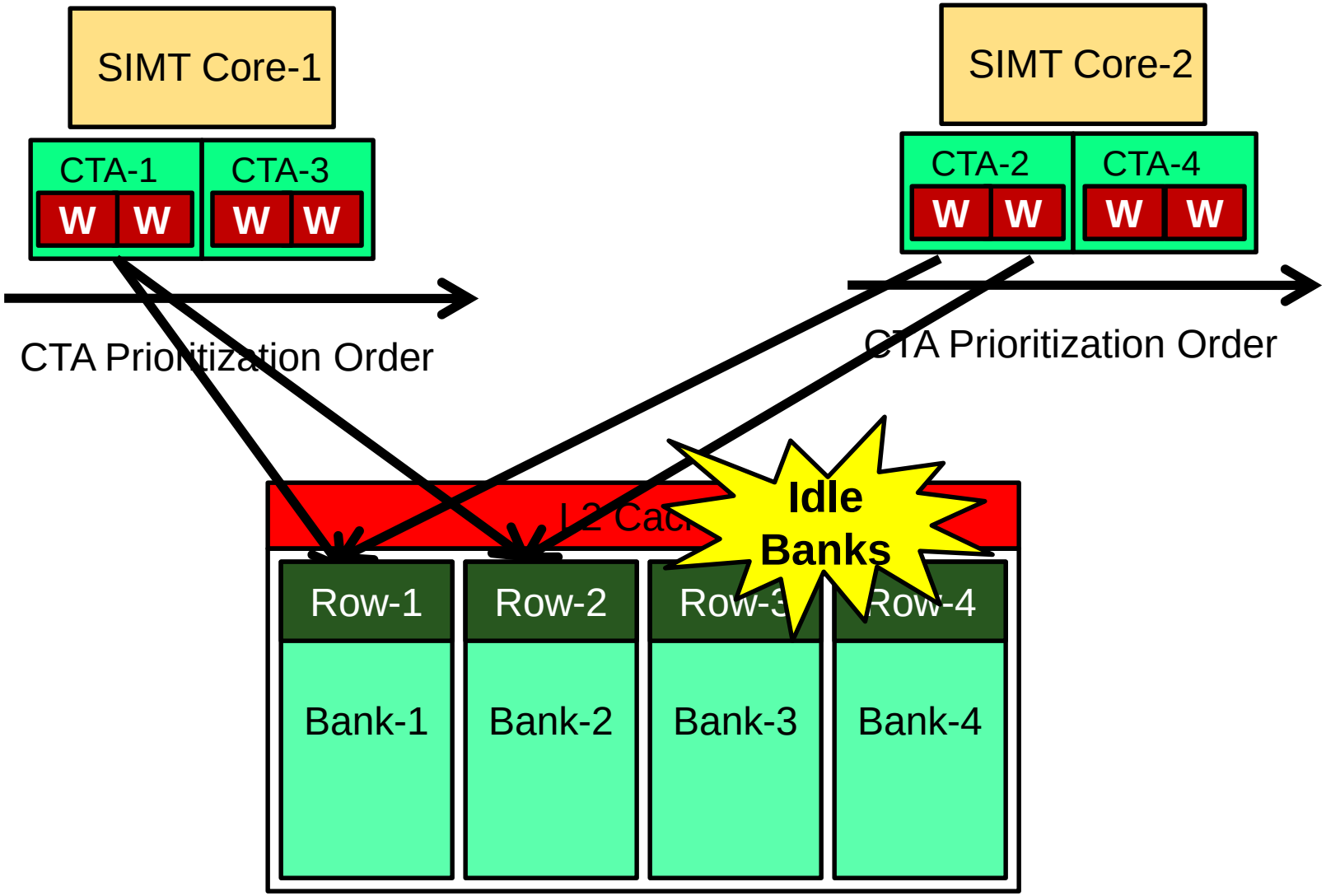
# CTA Data Layout (A Simple Example)



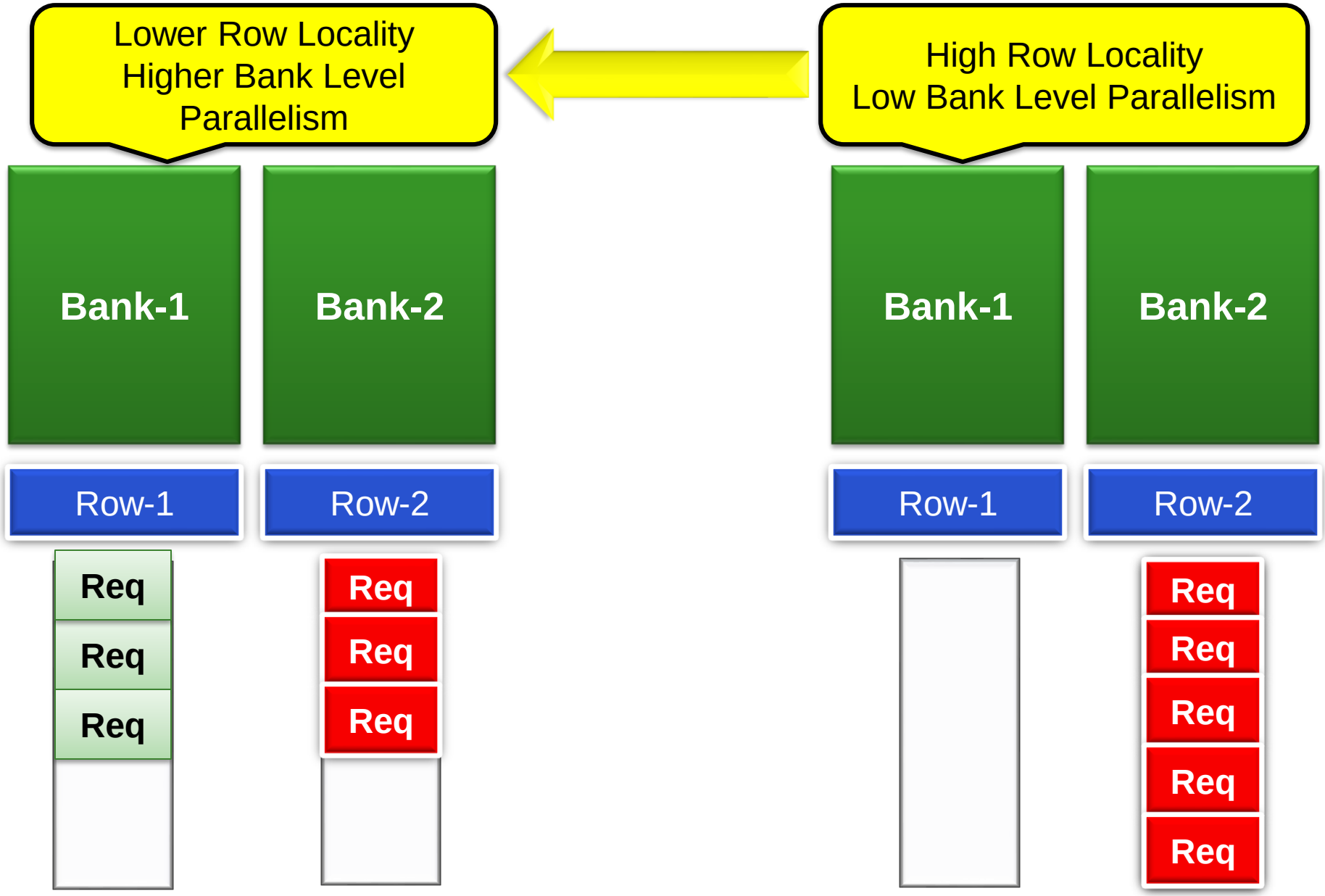

Average percentage of consecutive CTAs (out of total CTAs) accessing the same row = 64%

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Implications of high CTA-row sharing




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

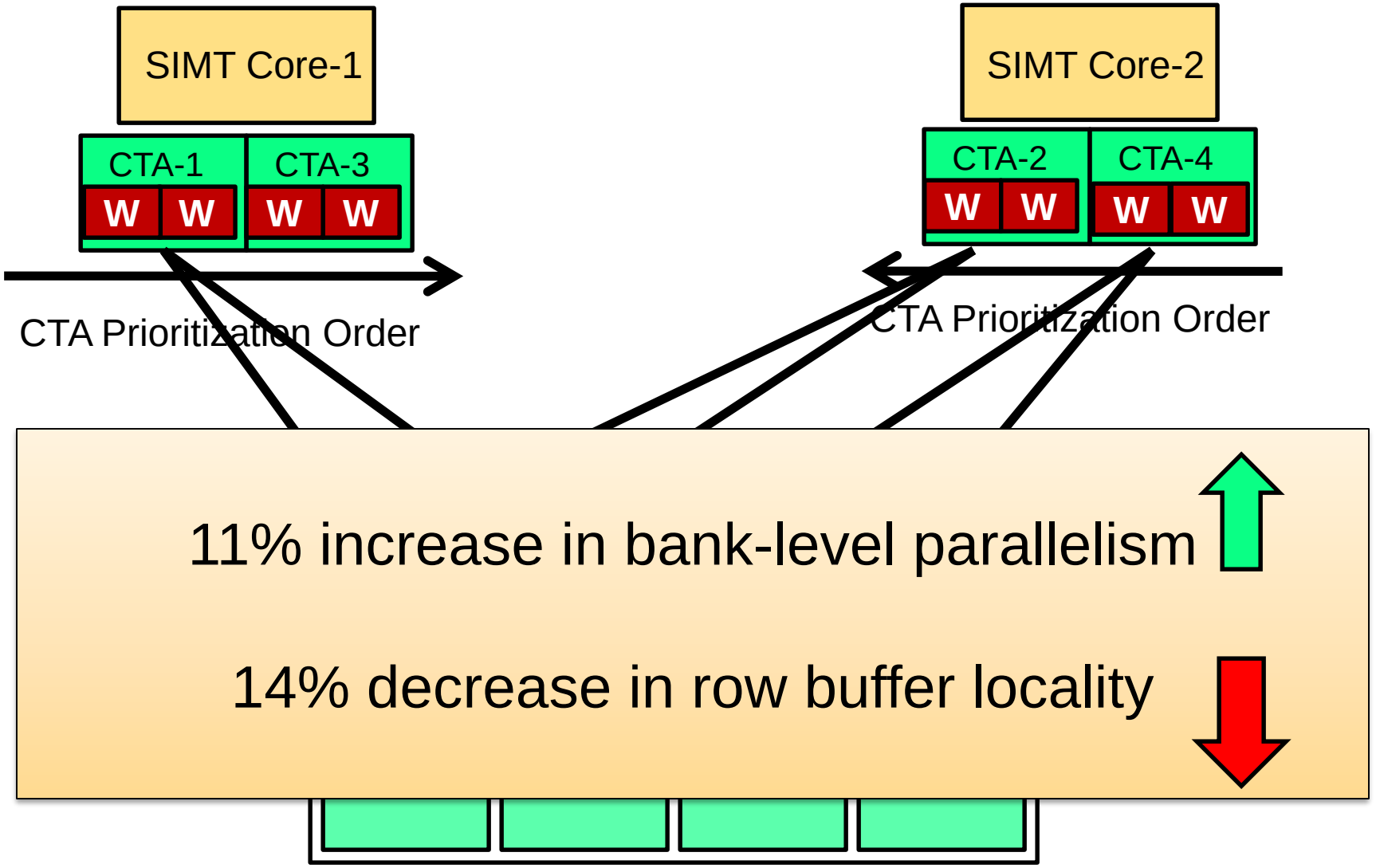



Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Some Additional Details

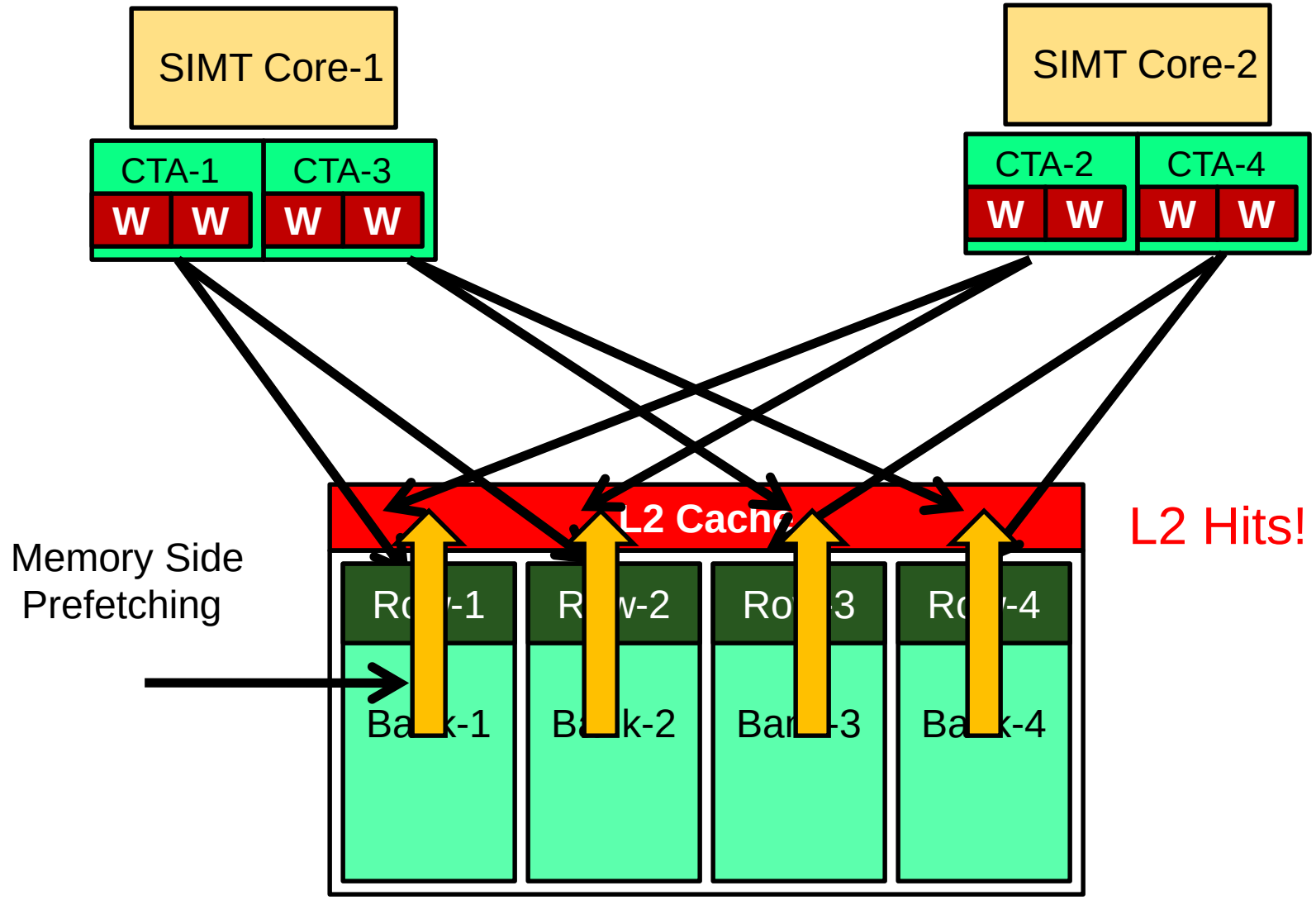
- Spread reference from multiple CTAs (on multiple SMs) across row buffers in the distinct banks
- Do not use same CTA group prioritization across SMs
  - Play  the odds
- What happens with applications with unstructured, irregular memory access patterns?

# Objective 2: Improving Bank Level Parallelism




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Objective 3: Recovering Row Locality




Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Memory Side Prefetching

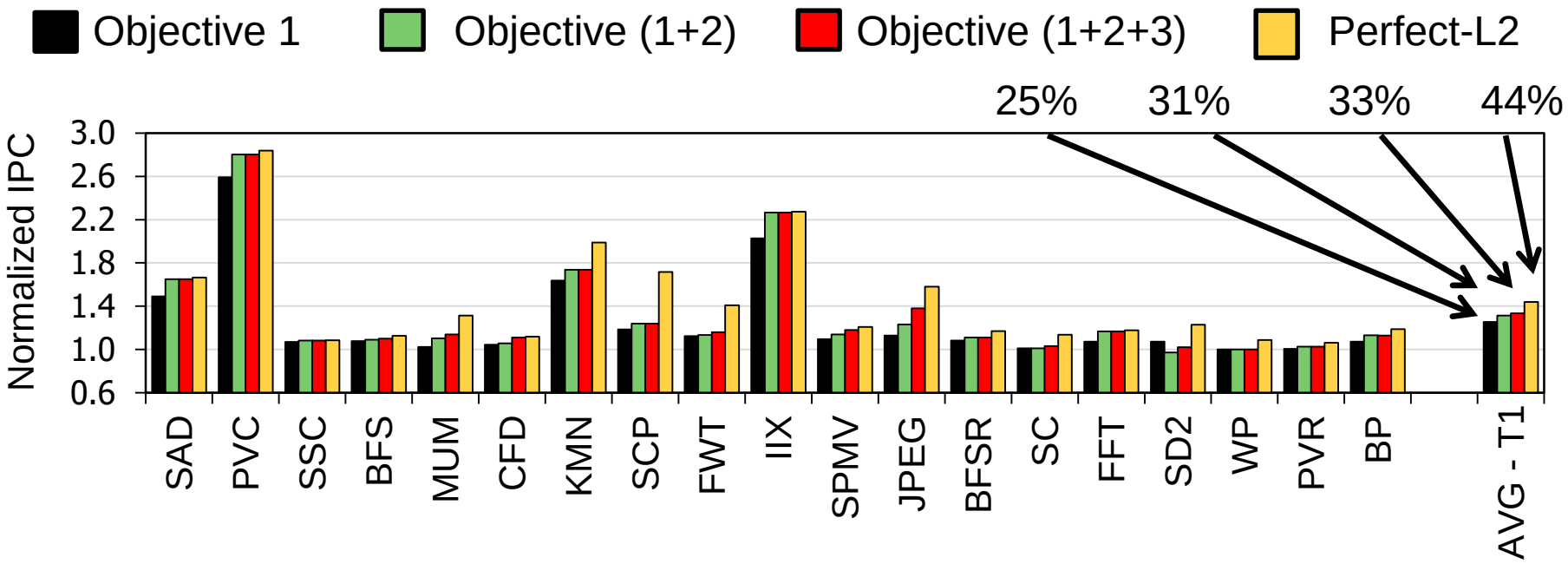
- Prefetch the so-far-unfetched cache lines in an already open row into the L2 cache, just before it is closed

- What to prefetch?
  - Sequentially prefetches the cache lines that were not accessed by demand requests
  - Sophisticated schemes are left as future work

- When to prefetch?
  - Opportunistic in Nature
  - Option 1: Prefetching stops as soon as demand request comes for another row. (Demands are always critical)
  - Option 2: Give more time for prefetching, make demands wait if there are not many. (Demands are NOT always critical)

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# IPC results (Normalized to Round-Robin)




- 11% within Perfect L2

Courtesy A. Jog, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, " ASPLOS 2013

# Summary

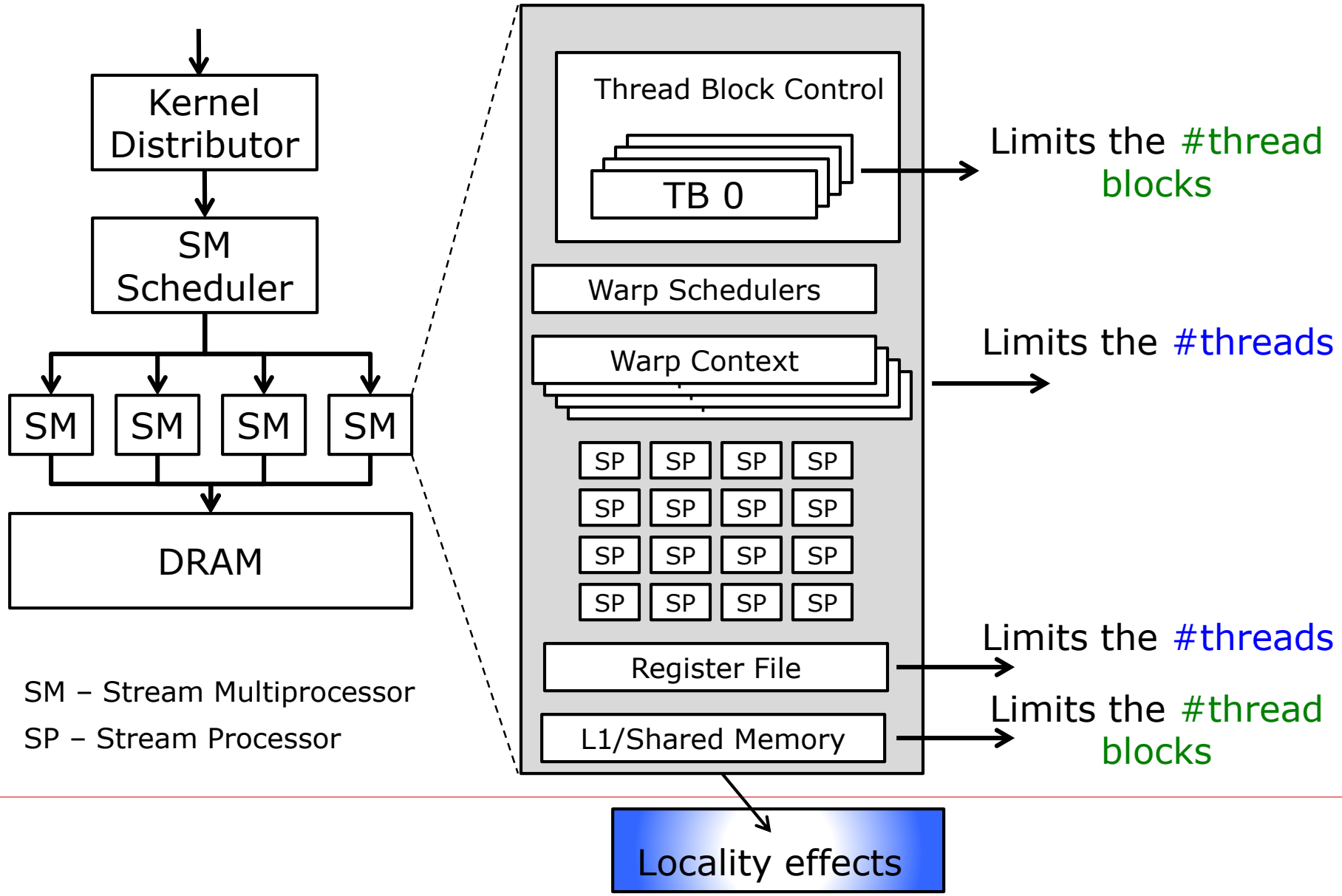
- Coordinated scheduling across SMs, CTAs, and warps
- Consideration of effects deeper in the memory system
- Coordinating warp residence in the core with the presence of corresponding lines in the cache

# CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration in GPGPU Workloads S. –Y Lee, A. A. Kumar and C. J Wu ISCA 2015

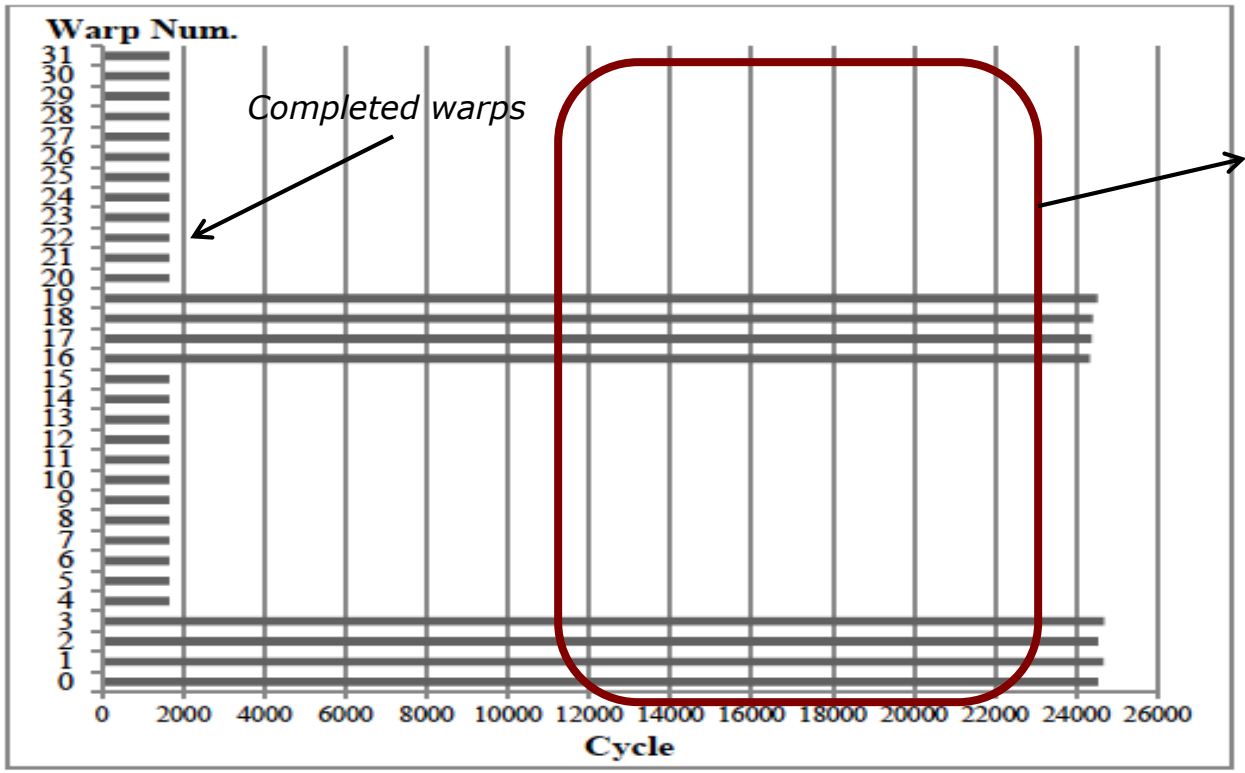
## Goal

- Reduce warp divergence and hence increase throughput
- The key is the identification of critical (lagging) warps
- Manage resources and scheduling decisions to speed up the execution of critical warps thereby reducing divergence

# Review: Resource Limits on Occupancy



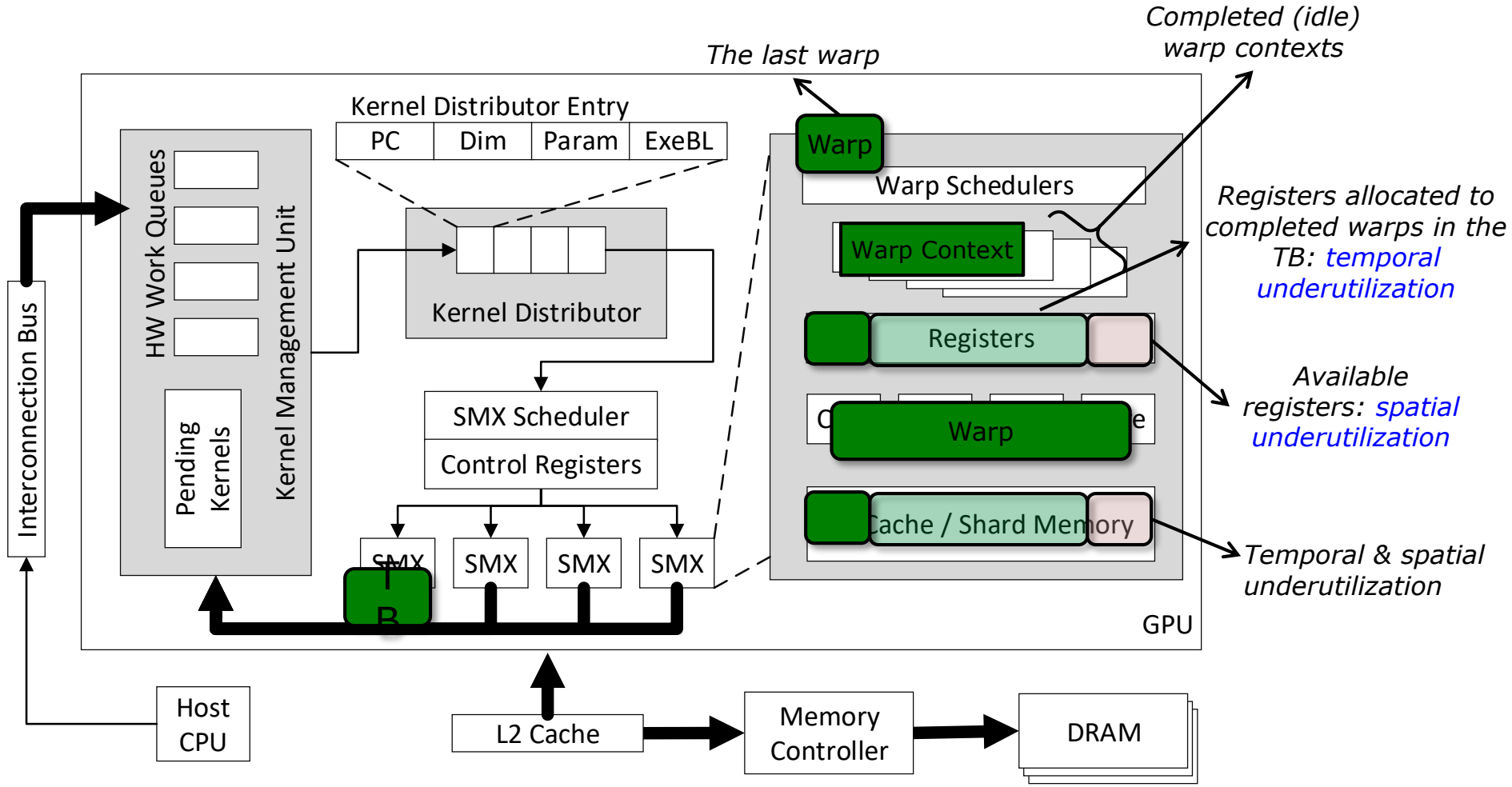
# Evolution of Warps in TB




- Coupled lifetimes of warps in a TB
  - Start at the same time
  - Synchronization barriers
  - Kernel exit (implicit synchronization barrier)

Figure from P. Xiang, Et. Al, " Warp Level Divergence: Characterization, Impact, and Mitigation

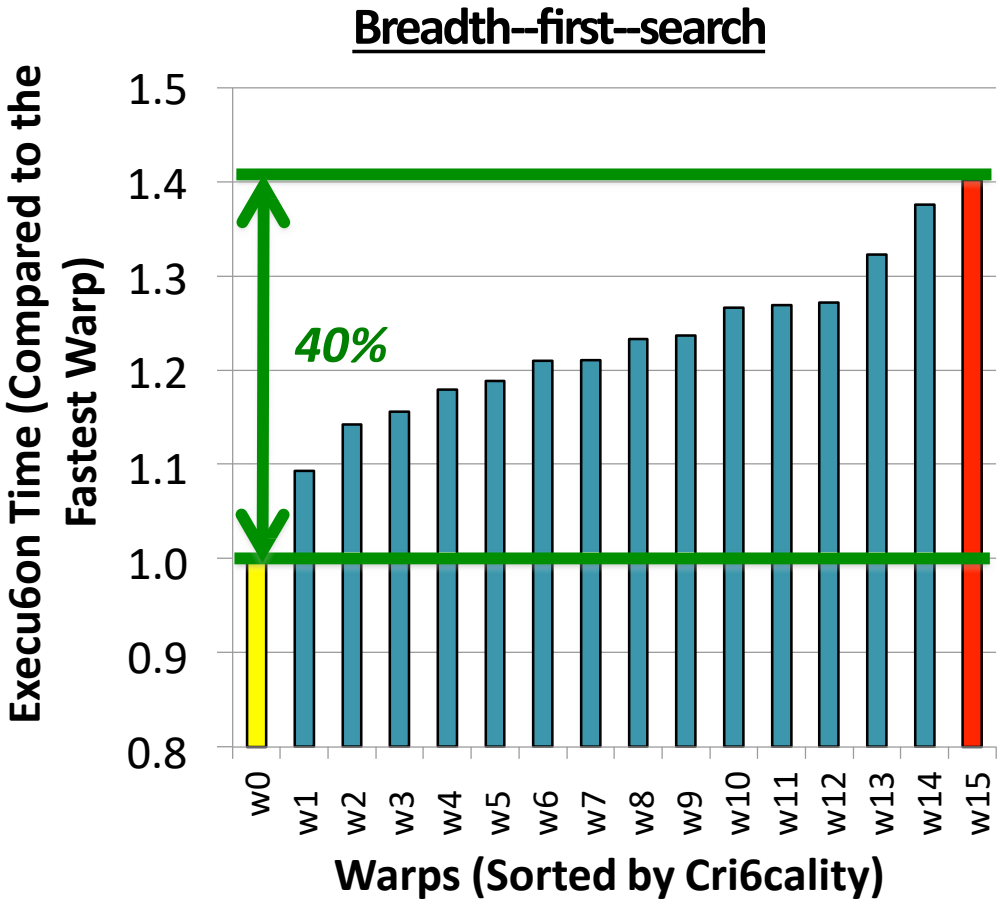
# Warp Criticality Problem

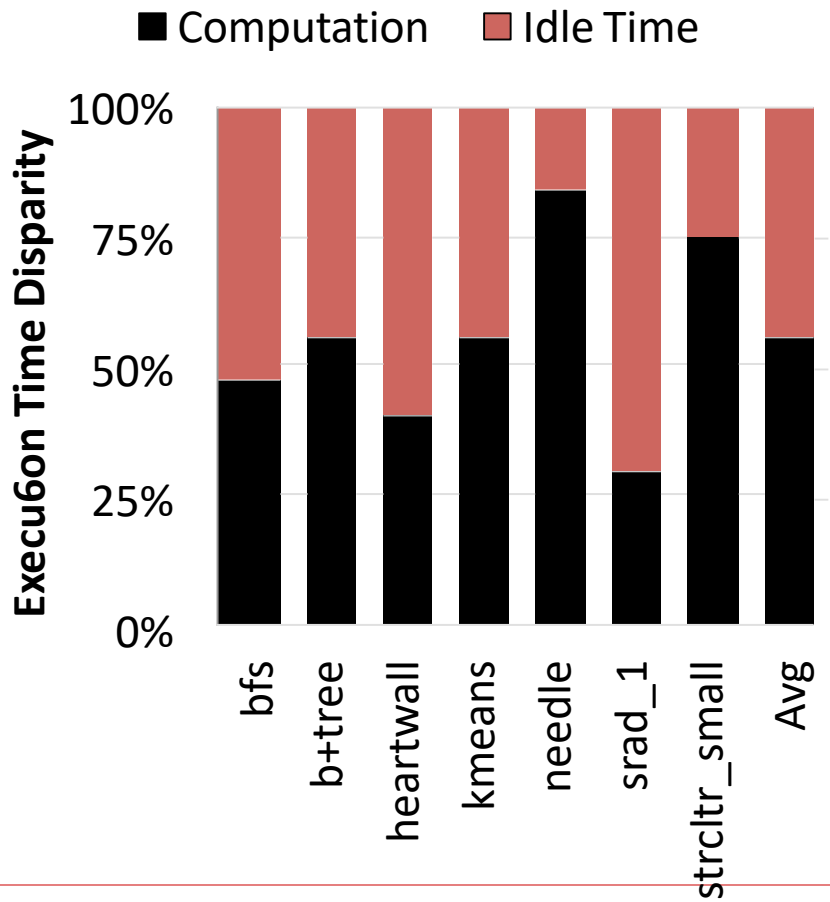



Manage resources and schedules around Critical Warps

# The Warp Criticality Problem

- Significant warp execution disparity for warps in the same thread block




*Lee and Wu, "*CAWS: Cri)cality--Aware Warp Scheduling for GPGPU Workloads*," PACT--2014

# Research Questions

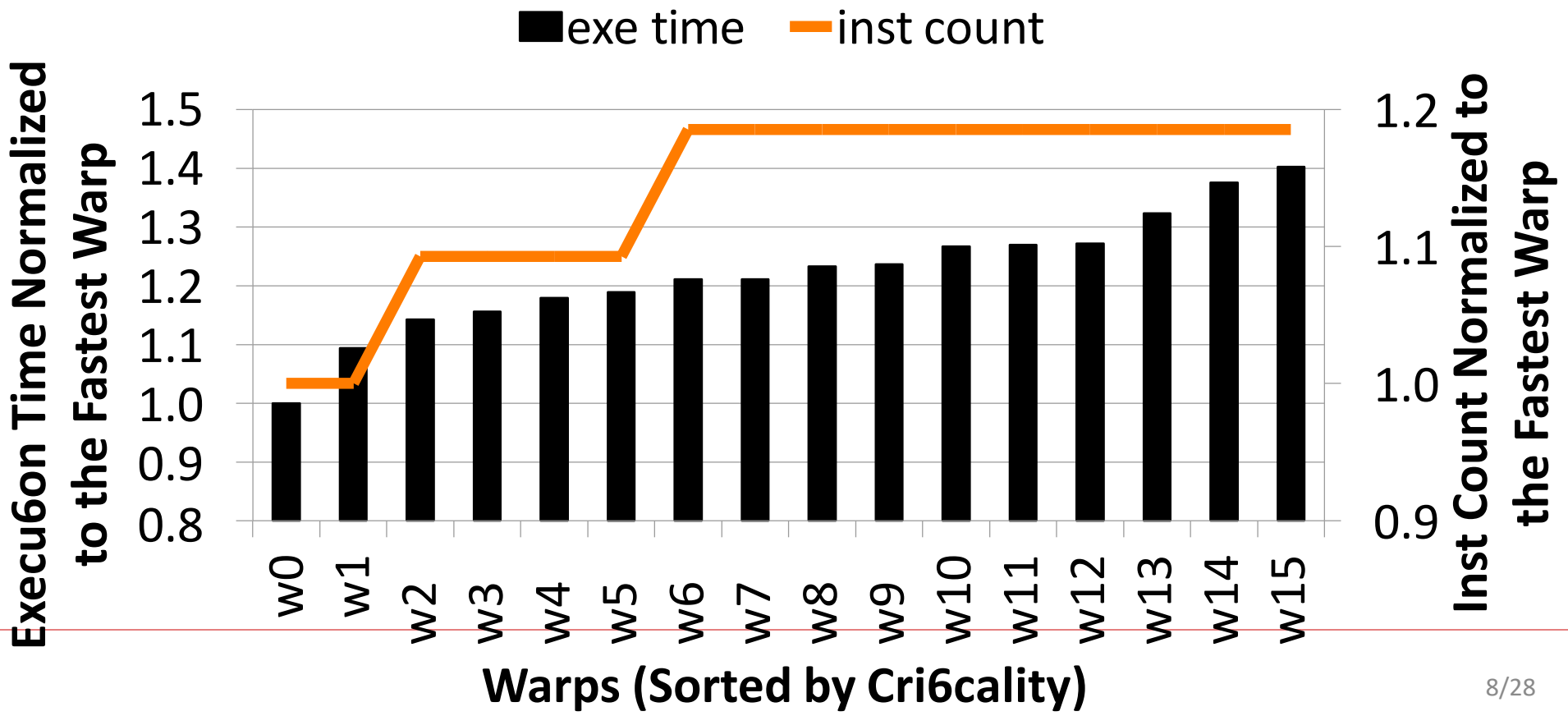
- What is the source of warp criticality?
- How can we effectively accelerate critical warp execution?

# Source of Warp Criticality

- Workload Imbalance
- Diverging Branch Behavior
- Memory Contention and Memory Access Latency
- Execution Order of Warp Scheduling

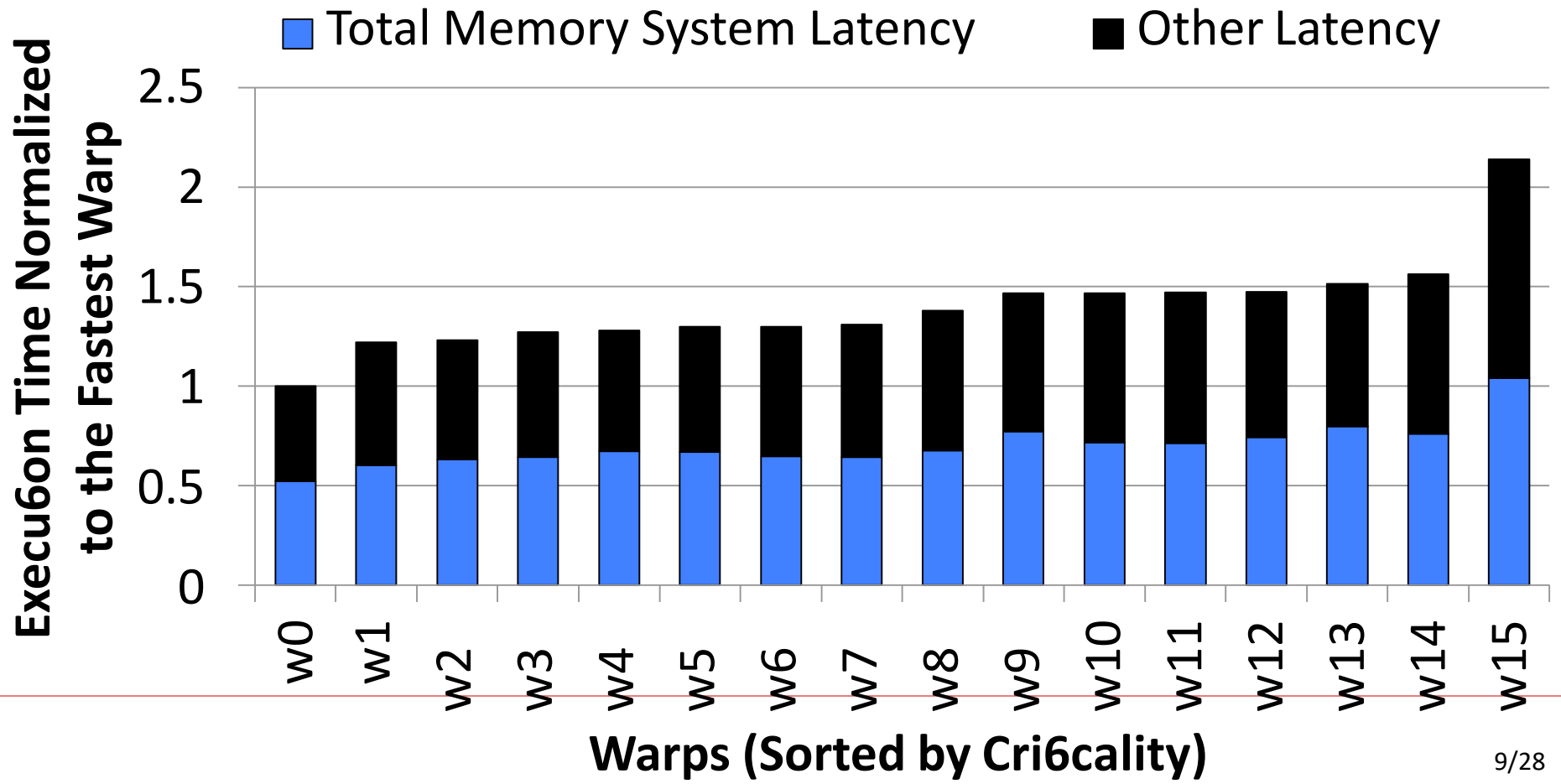
# Workload Imbalance & Diverging Branch

- Workload imbalance or diverging branch behavior makes warps have different number of dynamic instruction counts.



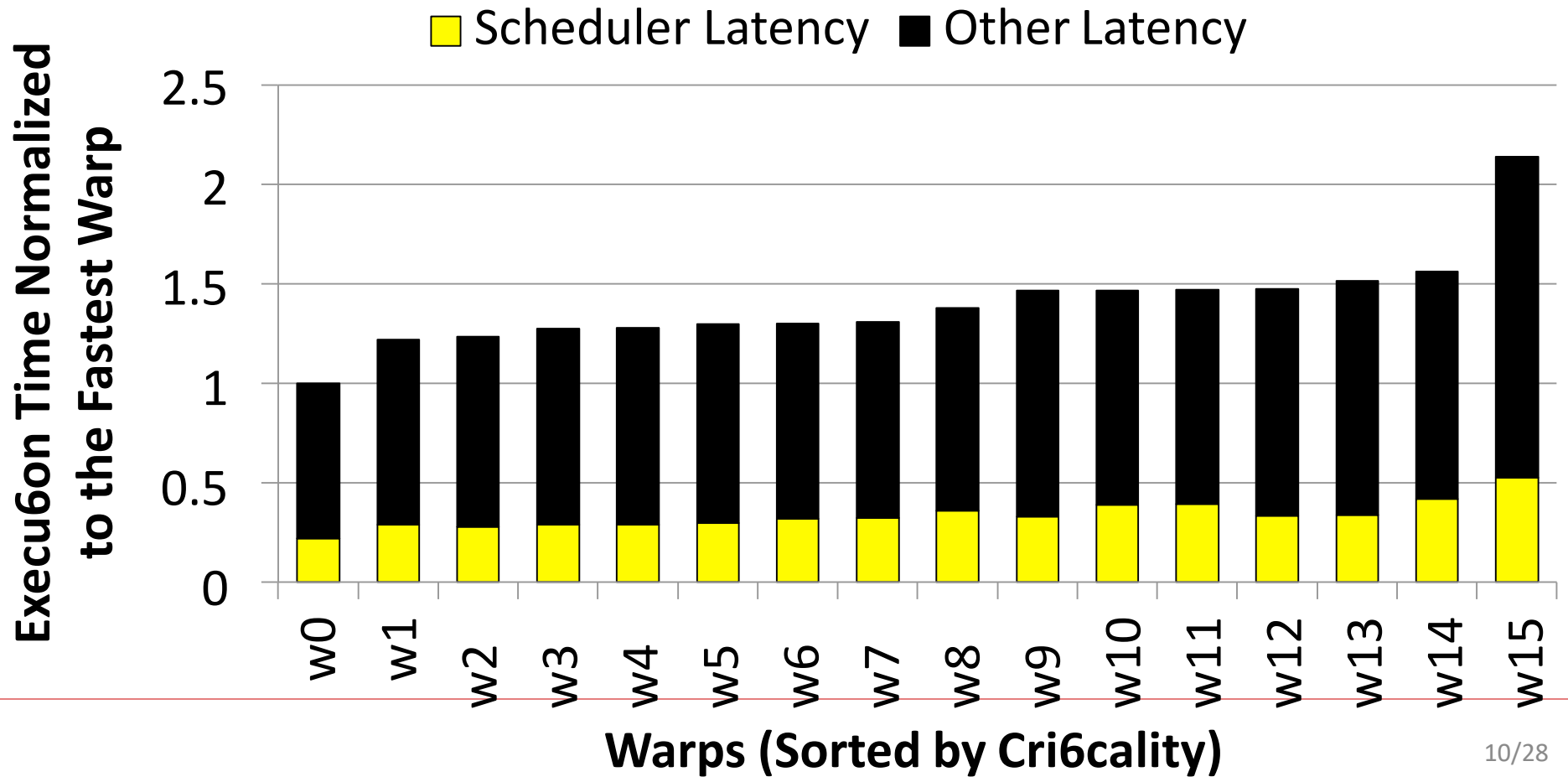
# Memory Contention

- While warps experience different latency to access memory, memory contention can induce warp criticality.



# Warp Scheduling Order

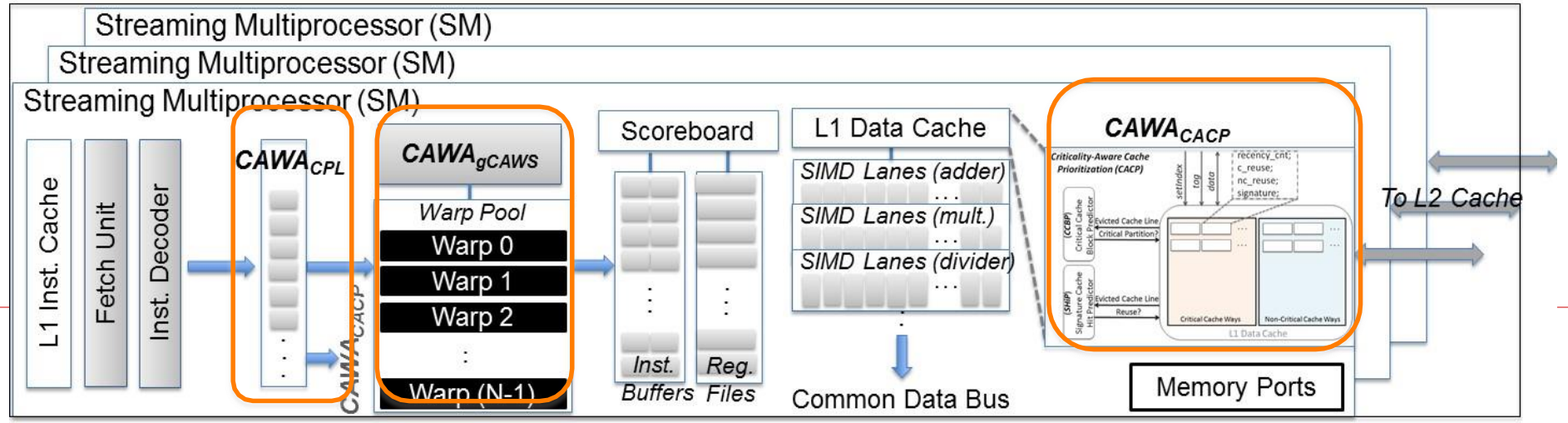
- The warp scheduler may introduce additional stall cycles for a ready warp, resulting in warp criticality



# CAWA: Criticality-Aware Warp Acceleration

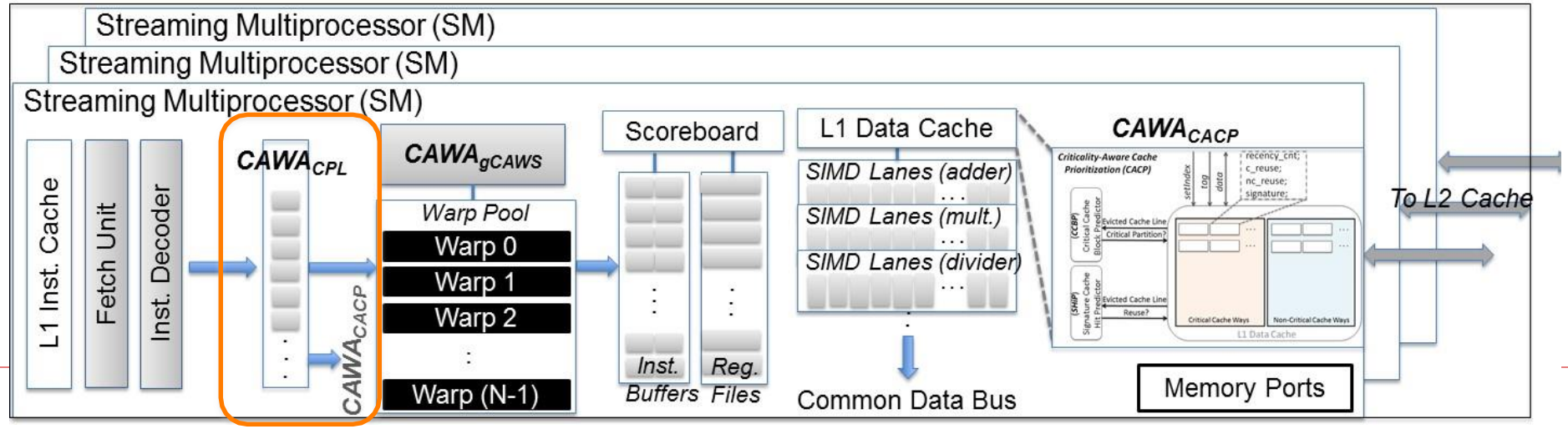
Coordinated warp scheduling and cache prioritization design

- Criticality Prediction Logic (CPL)
  - Predicting and identifying the critical warp at runtime
- greedy Criticality Aware Warp Scheduler (gCAWS)
  - Prioritizing and accelerating the critical warp execution
- Criticality--Aware Cache Prioritization (CACP)
  - Prioritizing and allocating cache lines for critical warp reuse



# CAWA: Criticality-Aware Warp Acceleration

- **Criticality Prediction Logic (CPL)**
  - **Predic6ng and iden6fying the cri6cal warp at run6me**
- greedy Criticality Aware Warp Scheduler (gCAWS)
  - Prioritizing and accelerating the critical warp execution
- Criticality--Aware Cache Prioritization (CACP)
  - Prioritizing and allocating cache lines for critical warp reuse



# $CAWA_{CPL}$ : Criticality Prediction Logic

- Evaluating number of additional cycles a warp may experience
- nInst is decremented whenever an instruction is executed

$$Criticality = nInst * w.CPIavg + nStall$$

instruction count disparity
diverging branch

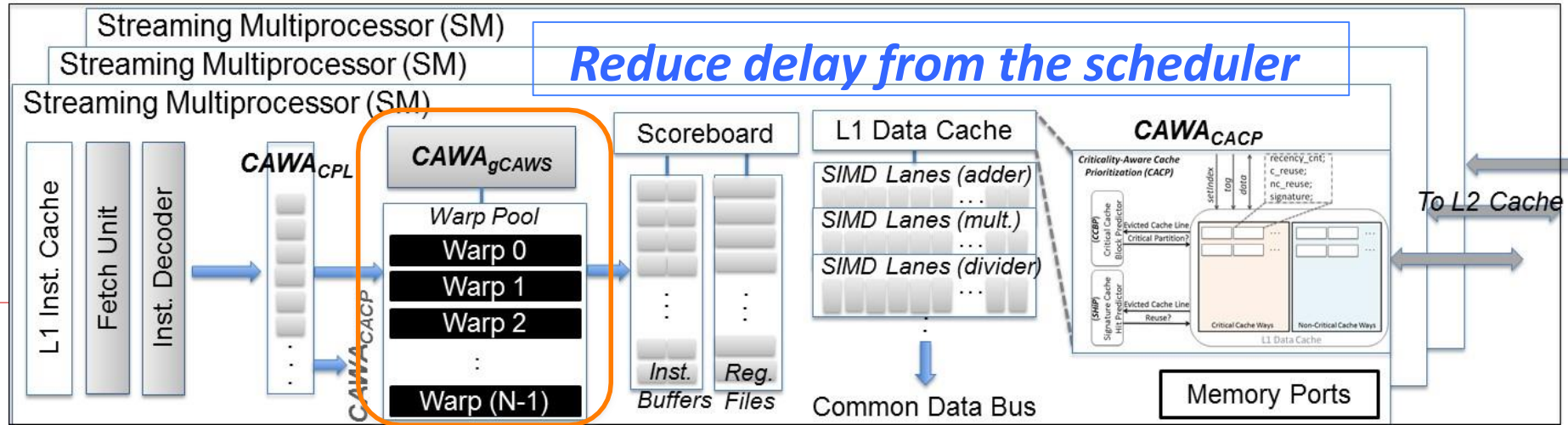
memory latency
scheduling latency

```
if(……)
      inst-1
      …
      inst-m
else
      inst-1
      …
      inst-n
```

| | Branch path | Per-warp #inst. |
|---|---|---|
| *Yes:* thread branch divergence | | m+n |
| *No:* thread branch divergence | not-taken | m |
| | taken | n |

# CAWA: Criticality-Aware Warp Acceleration

- Criticality Prediction Logic (CPL)
  - Predicting and identifying the critical warp at runtime
- **greedy Criticality Aware Warp Scheduler (gCAWS)**
  - **Prioritizing and accelerating the critical warp execu6on**
- Criticality--Aware Cache Prioritization (CACP)
  - Prioritizing and allocating cache lines for critical warp reuse



# $CAWA_{gCAWS}$ : greedy Criticality-Aware Warp Scheduler

- • Prioritizing warps based on their criticality given by gCAWS
- • Executing warps in a greedy* manner
- – Select the most critical ready--warp
- – Keep on executing the select warp until it stalls

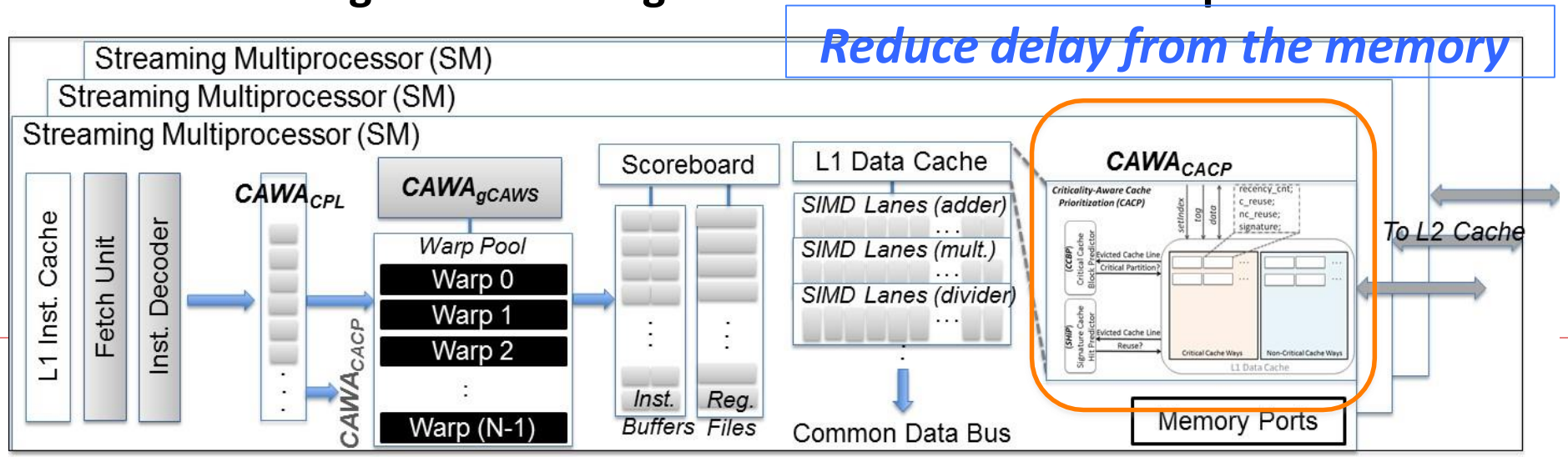
| Warp Pool | Criticality |
|---|---|
| Warp 0 | 5 |
| Warp 1 | 10 |
| Warp 2 | 3 |
| Warp 3 | 7 |

- **Warp Scheduler Selec6on Sequence**
- Traditional Approach (e.g. RR, 2L, GTO):
- *W0→W1→W2→W3*
  - gCAWS:
- *W1→W3→W0→W2*

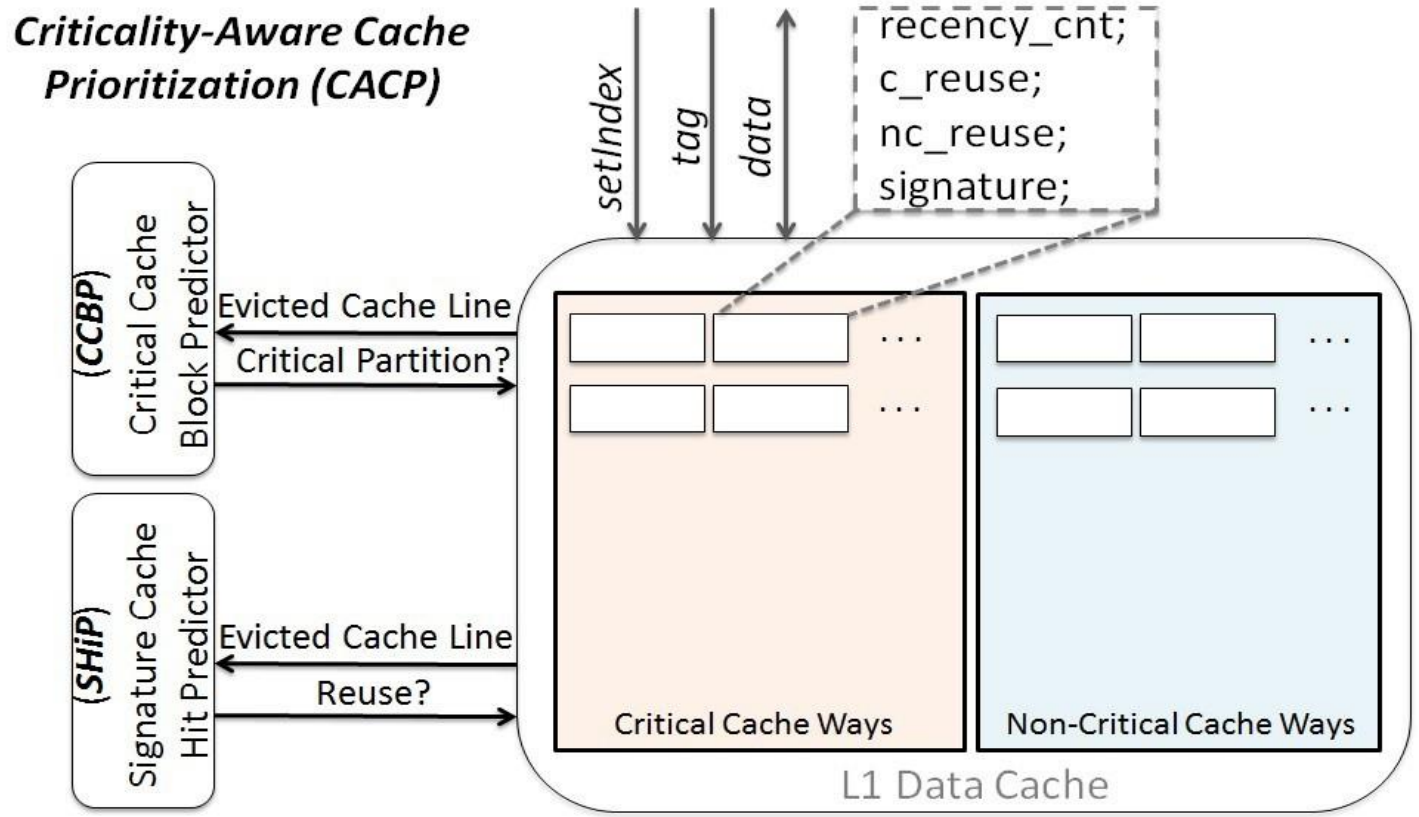
*Rogers et al., "*Cache--Conscious Wavefront Scheduler*," MICRO'12

# CAWA: Criticality-Aware Warp Acceleration

- Criticality Prediction Logic (CPL)
  - Predicting and identifying the critical warp at runtime
- greedy Criticality Aware Warp Scheduler (gCAWS)
  - Prioritizing and accelerating the critical warp execution
- **Cri6cality--Aware Cache Priori6za6on (CACP)**
  - **Priori6zing and alloca6ng cache lines for cri6cal warp reuse**

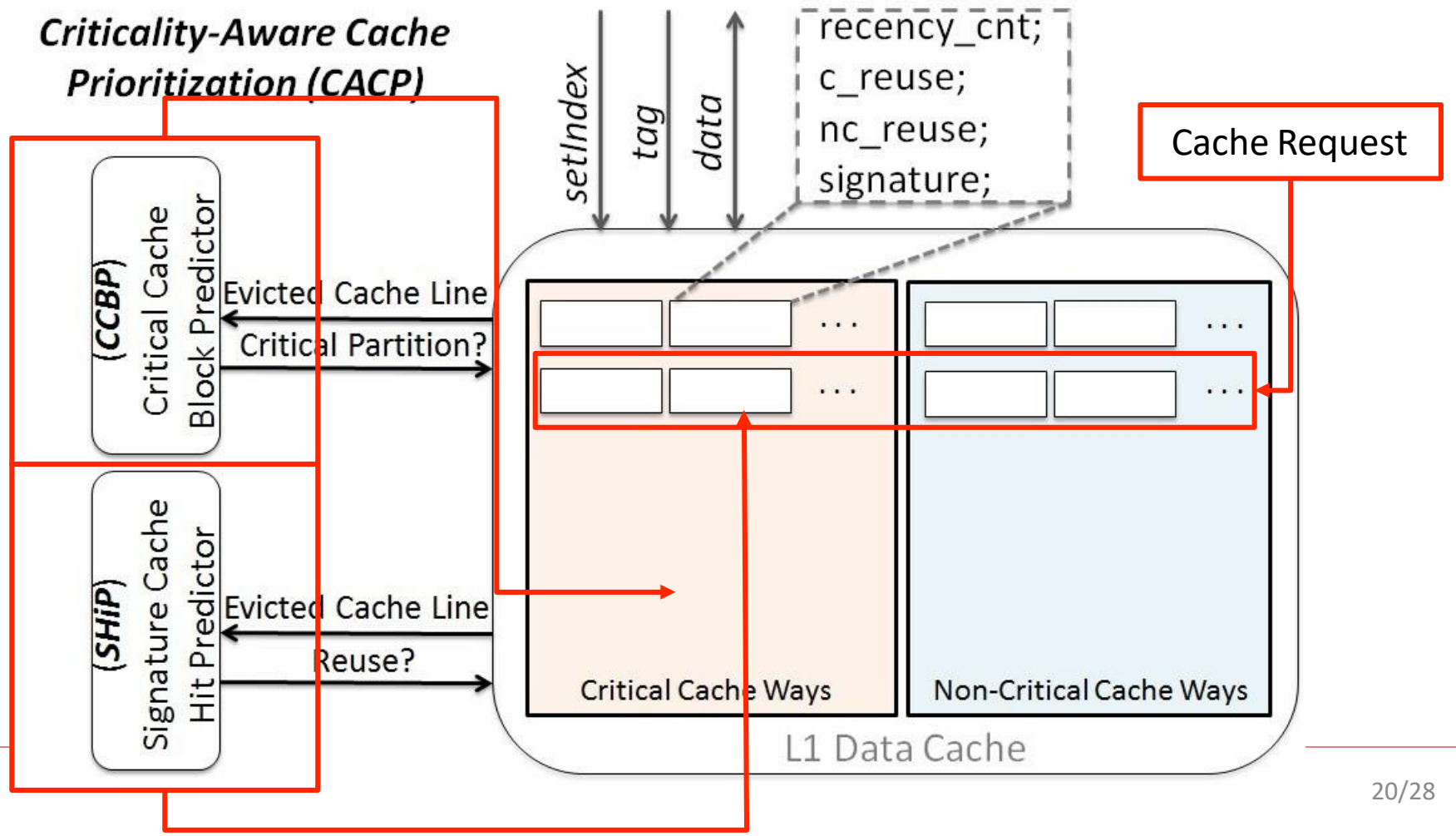


# $CAWA_{CACP}$ : Criticality-Aware Cache Prioritization

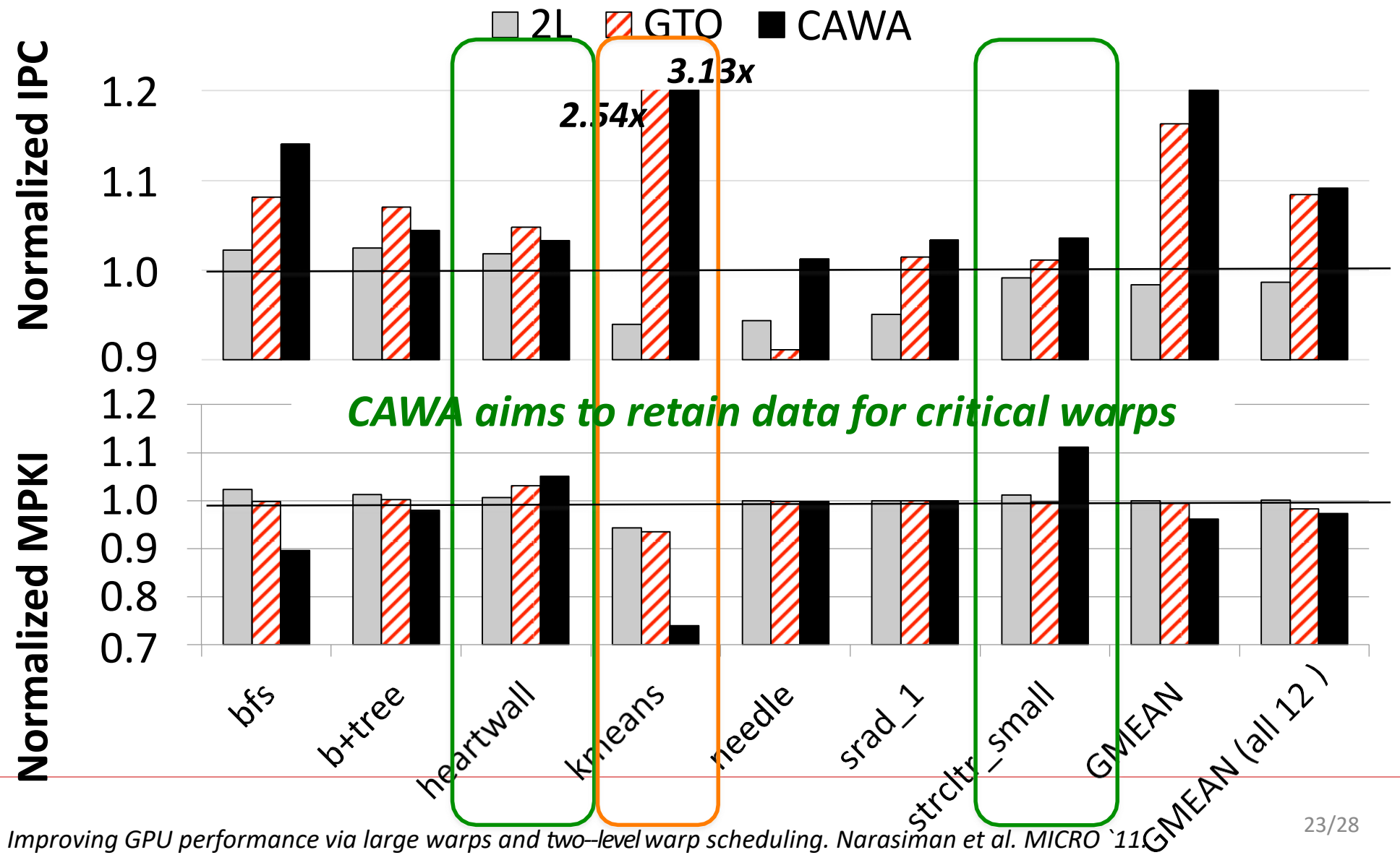



*Wu et al., "*SHiP: Signatured--based Hit Predictor for High Performance Caching*," MICRO'11

# $CAWA_{CACP}$ : Criticality-Aware Cache Prioritization

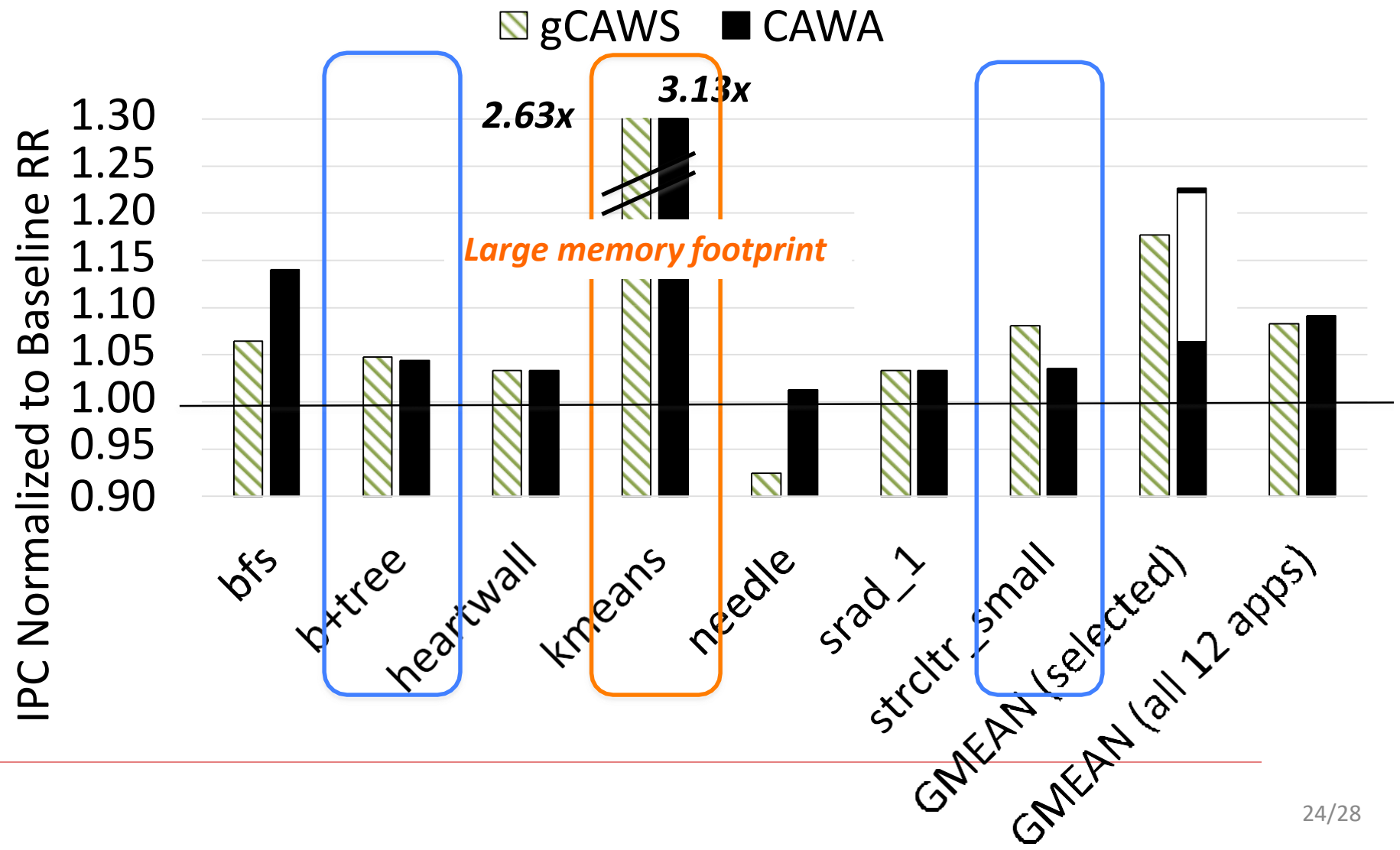


# Overall Performance Improvement

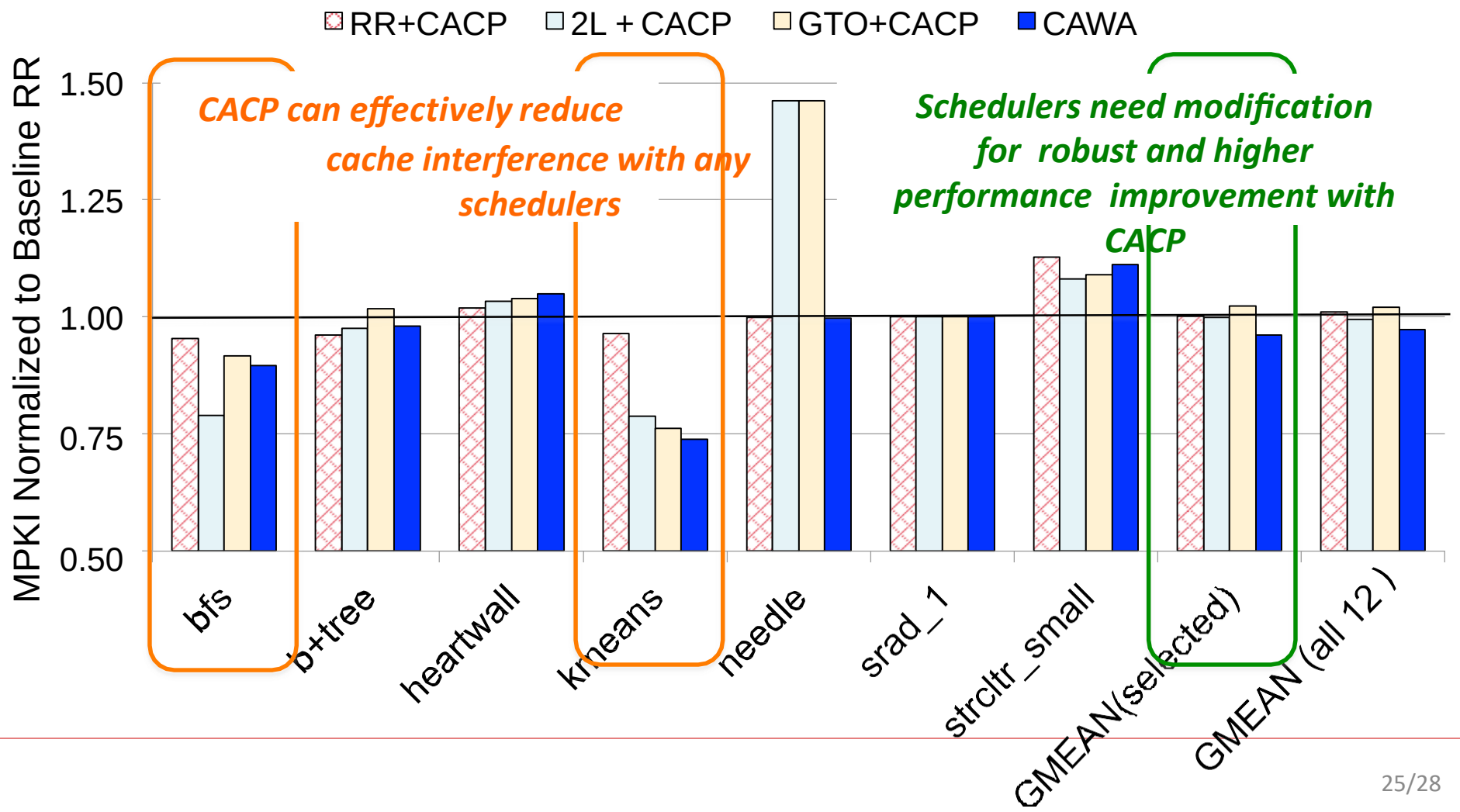



*Improving GPU performance via large warps and two–level warp scheduling. Narasiman et al. MICRO `11.*
*Cache conscious wavefront scheduling. Rogers, O'Connor, and Aamodt. MICRO `12.*

# gCAWS Performance Improvement



# Performance Improvement with $CAWA_{CACP}$



# Summary

- Warp divergence leads to some lagging warps → critical warps
- Expose the performance impact of critical warps → throughput reduction
- Coordinate scheduler and cache management to reduce warp divergence