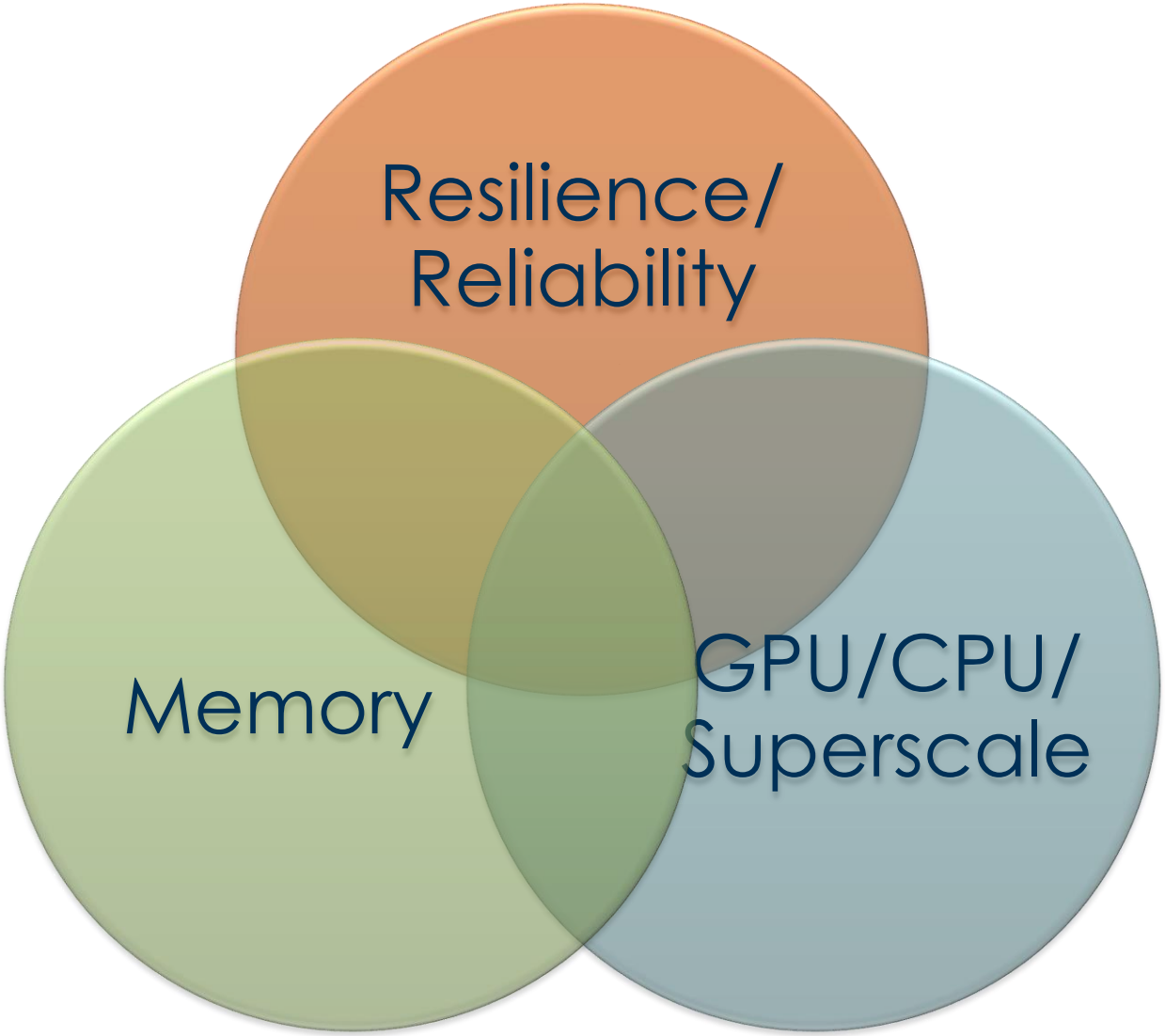
Robust GPU Architectures

Improving Irregular Execution on Architectures Tuned for Regularity

Mattan Erez

The University of Texas at Austin

Lots of interesting multi-level projects



Arch-focused whole-system approach

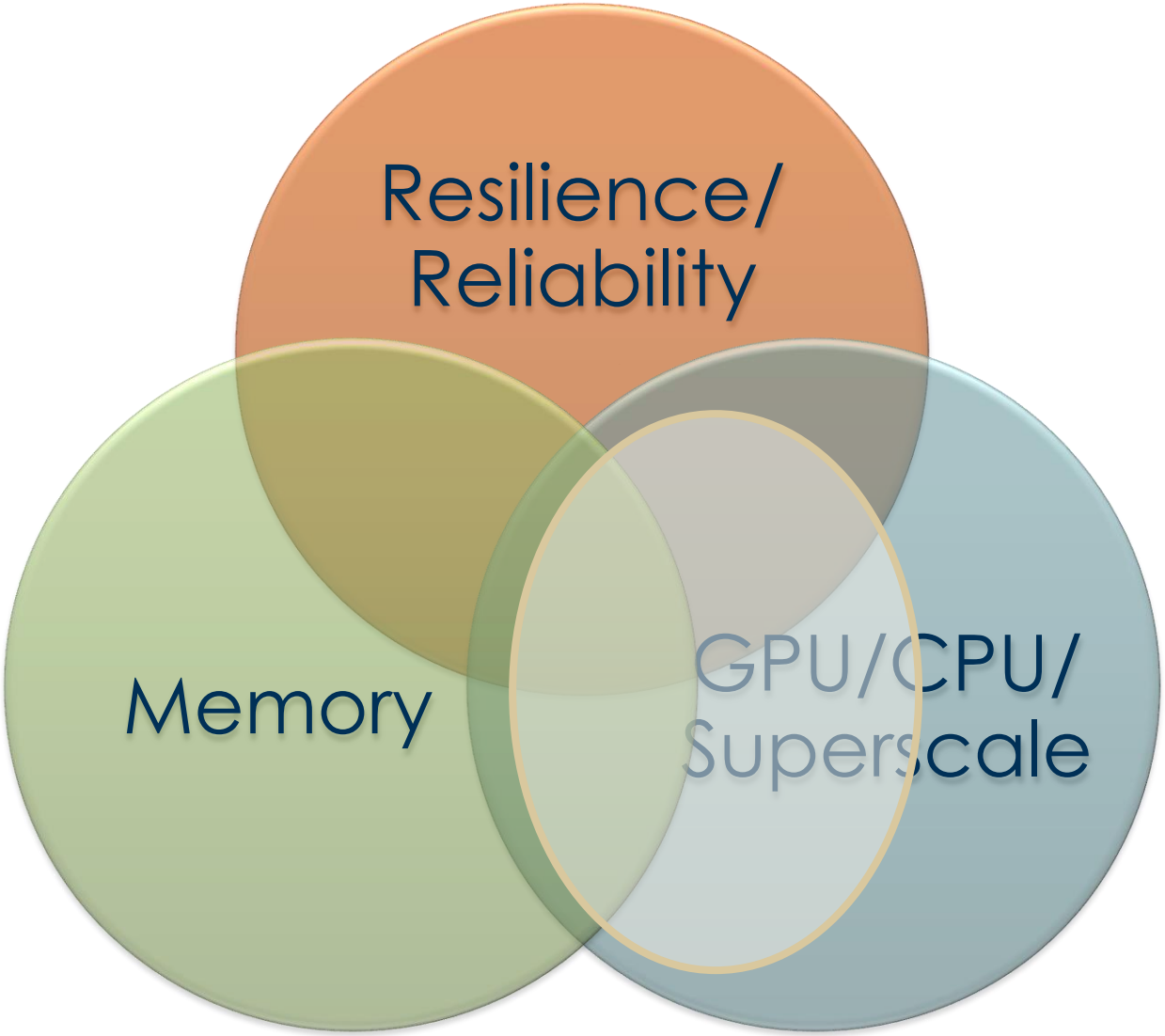
- Efficiency requirements require crossing layers
- Algorithms are key
  - Compute less, move less, store less
- Proportional systems
  - Minimize waste
- Utilize and improve emerging technologies

- Explore (and act) up and down
  - Programming model to circuits
- Preferably implement at micro-arch/system

Big problems and emerging platforms

- Memory systems
  - Capacity, bandwidth, efficiency – impossible to balance
  - Adaptive and dynamic management helps
  - New technologies to the rescue?
  - Opportunities for in-memory computing
- GPUs, clouds, and more
  - Throughput oriented designs are a must
  - Centralization trend is interesting
- Reliability and resilience
  - More, smaller devices – danger of poor reliability
  - Trimmed margins – less room for error
  - Hard constraints – efficiency is a must
  - Scale exacerbates reliability and resilience concerns

Lots of interesting multi-level projects



Regularity is good for hardware

- Regularity exploited to amortize overheads
- SIMD/vectors

- Long(ish) cache lines and memory transfers

Good algorithms often irregular

- Graphs
- Sparse structures
- Early-exit conditions
- Specialization
- …

What to do?

- Tune and regularize algorithm implementation
  - Works great
  - Hard to do
  - Sometimes impossible
- Build hardware for irregularity
  - Fine-grained control and access
  - Need to make it cheap enough
- Need to not penalize regular(ized) algorithms!
- Need a **robust architecture**

Dynamic adaptivity and flexibility

- Hardware mostly designed for regularity
  - Negligible impact on regular portions
- Low-overhead structures support irregularity
  - Performance and efficiency gains on irregular parts

- Improving irregular control on wide-SIMD GPUs
  - Dynamically “regularize” diverged branches
  - Utilize divergence to increase TLP
- Improving fine-grain memory access
  - Locality-aware memory hierarchy
  - Dynamic granularity accesses

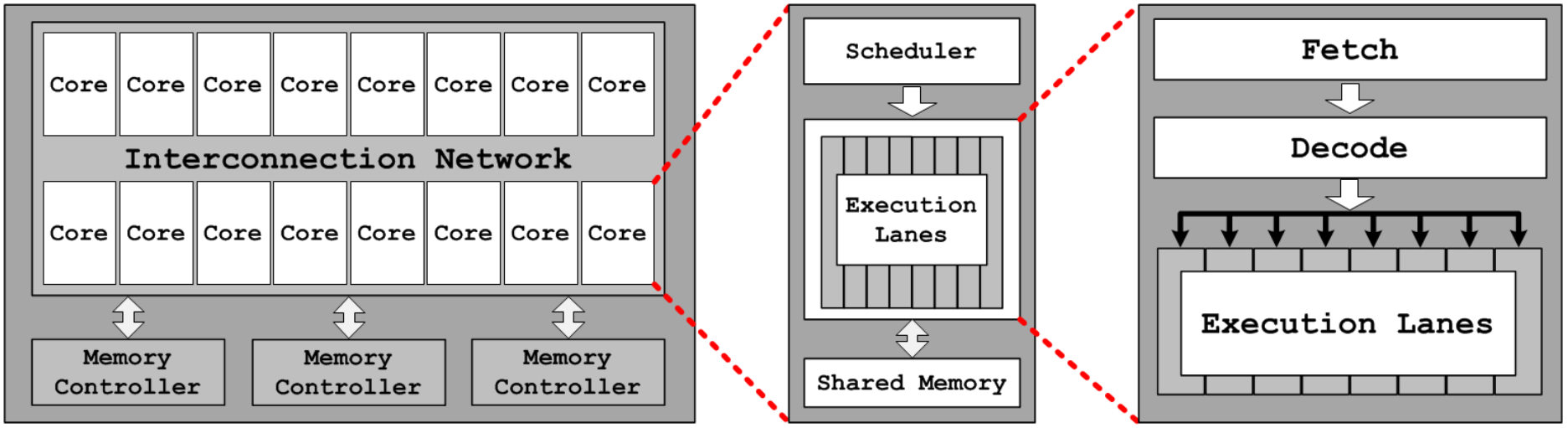
Outline

- Robust control for wide-SIMD GPUs
  - Brief "GPU" overview
  - Dynamically "regularize" diverged branches
    - Warp compaction
    - Eliminating its impact on regular code
  - Utilize divergence to increase TLP
    - Dual-path execution
- Robust memory access
  - Dynamic granularity accesses
  - Locality-aware memory hierarchy for GPUs
  - Robust caching
- **The real credit belongs to the students**
  - **Min Kyu Jeong, Minsoo Rhu, Michael Sullivan, Doe Hyun Yoon, Dong Li**

BRIEF GPU OVERVIEW

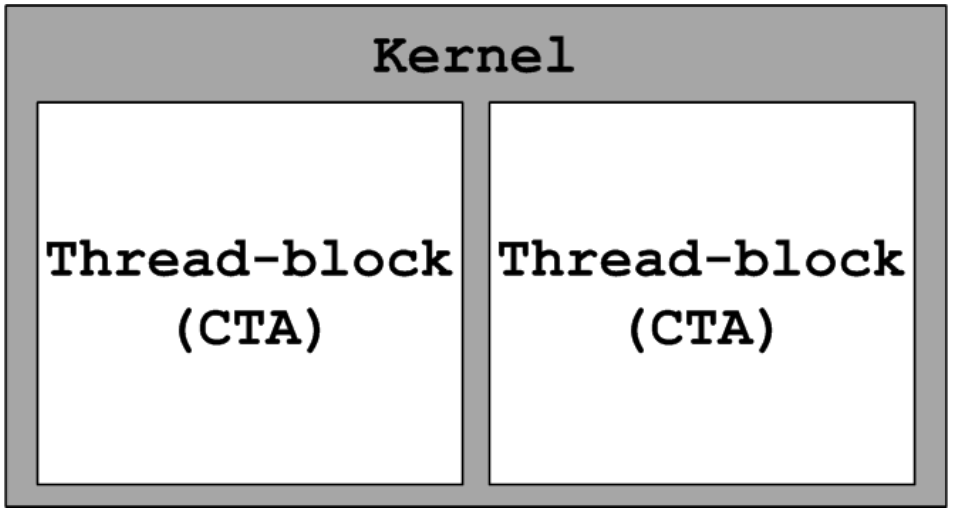
Graphic Processing Units (GPUs)

- General-purpose many-core accelerators
  - Use 100s of simple in-order shader cores
  - Shader core == **s**treaming **m**ultiprocessor (**SM**) in NVIDIA GPUs
- Scalar frontend (fetch & decode) + parallel backend
  - Amortize the cost of frontend and control



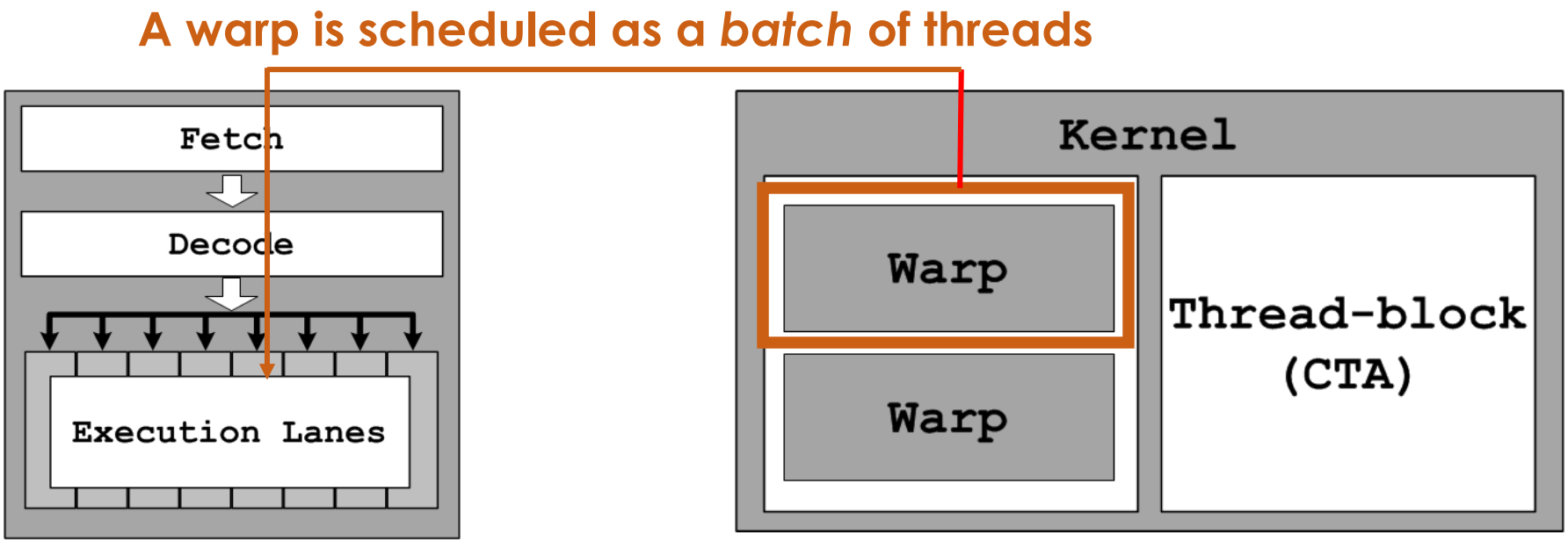
Exposed hierarchy of data-parallel threads

- SPMD: single ***kernel*** executed by numerous ***scalar*** threads
- Kernel / Thread-block hierarchy
  - Kernel composed of many **thread-blocks** (a.k.a. cooperative-thread-arrays (CTAs) or work-groups)



CUDA exposes hierarchy of data-parallel threads

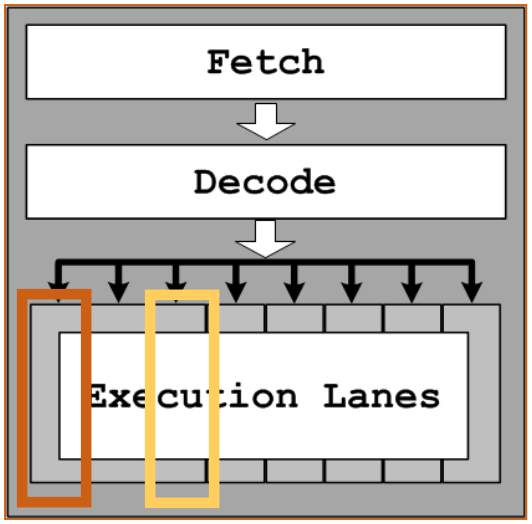
- SPMD: single ***kernel*** executed by numerous ***scalar*** threads
- Kernel / Thread-block / *Warp* / *Thread*
  - Multiple **warps** compose a **thread-block** (OpenCL wavefronts)
  - Multiple **threads (32)** compose a warp (OpenCL work-items)

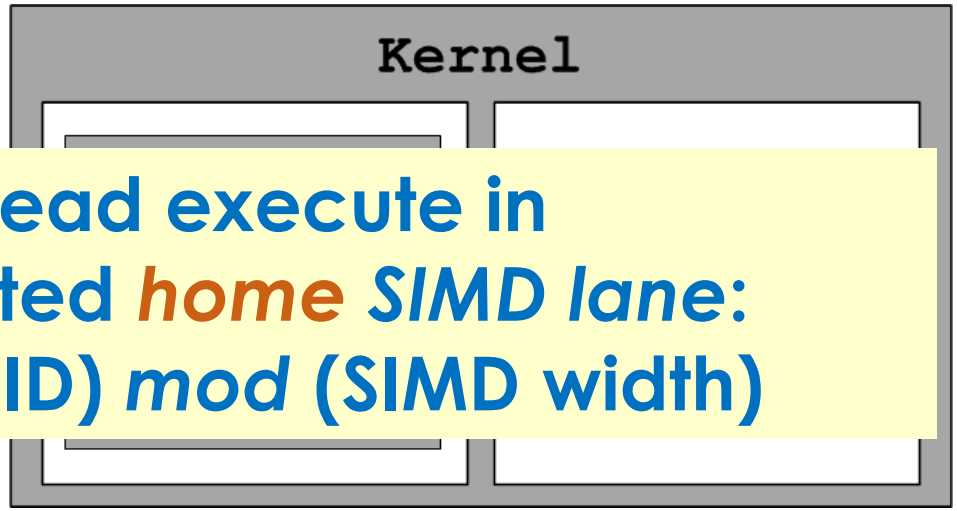


CUDA exposes hierarchy of data-parallel threads

- SPMD: single *kernel* executed by numerous *scalar* threads
- Kernel / Thread-block / *Warp* / *Thread*
  - Multiple **warps** compose a **thread-block**
  - Multiple **threads (32)** compose a warp

: Thread-ID 0, 32, 64 … execute in physical lane #0
: Thread-ID 2, 34, 66 … execute in physical lane #2

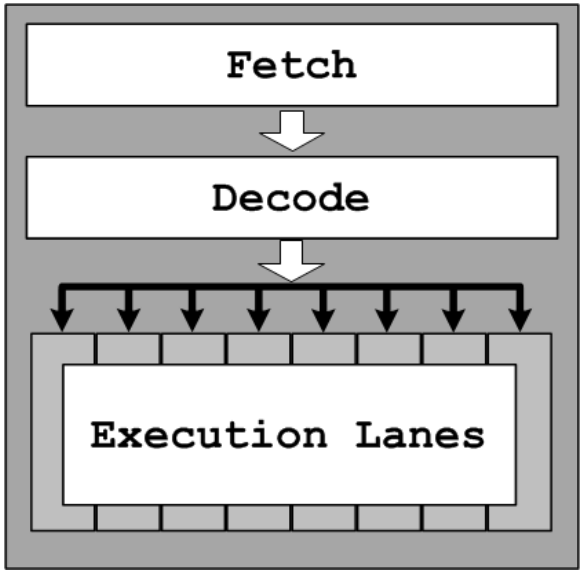




Each thread execute in designated *home SIMD lane*: (thread-ID) *mod* (SIMD width)

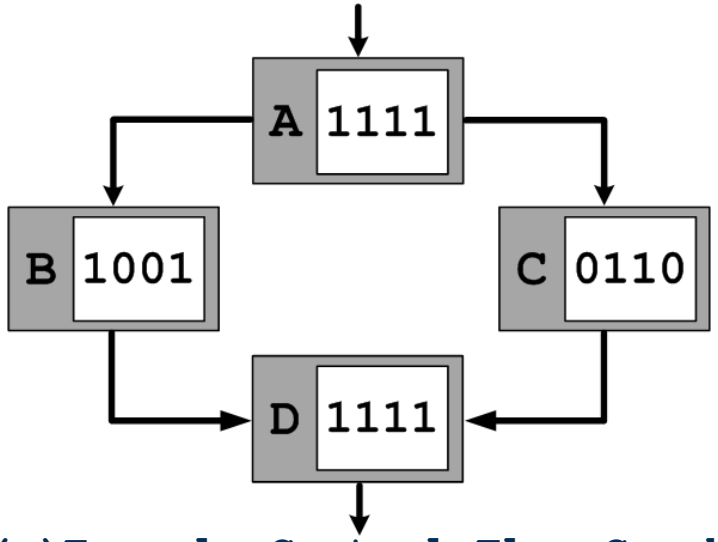
GPUs have HW support for conditional branches

- SIMT: Single-Instruction Multiple-Thread
  - Underlying hardware uses ***vector*** SIMD pipelines (lanes)
  - Programmer writes code to be executed by ***scalar*** threads
  - Hardware/software supports conditional branches
    - Each thread can follow its own control flow

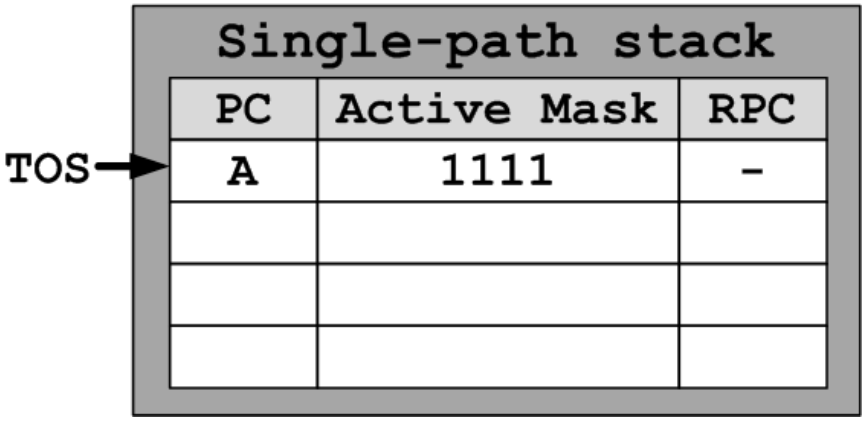


SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute *active*-threads at the ***top-of-stack*** (TOS)
  - **Active bitmasks** of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at *reconvergence point*
  - **Reconvergence PC** (RPC) point derived at compile-time
    - RPC: ***immediate post-dominator (PDOM)*** of the branching-point



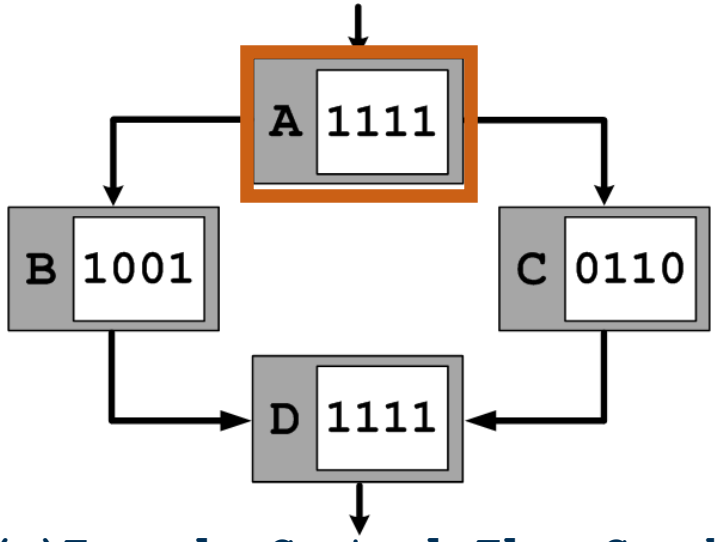
(a) Example Control Flow Graph
1: Active, 0: Inactive



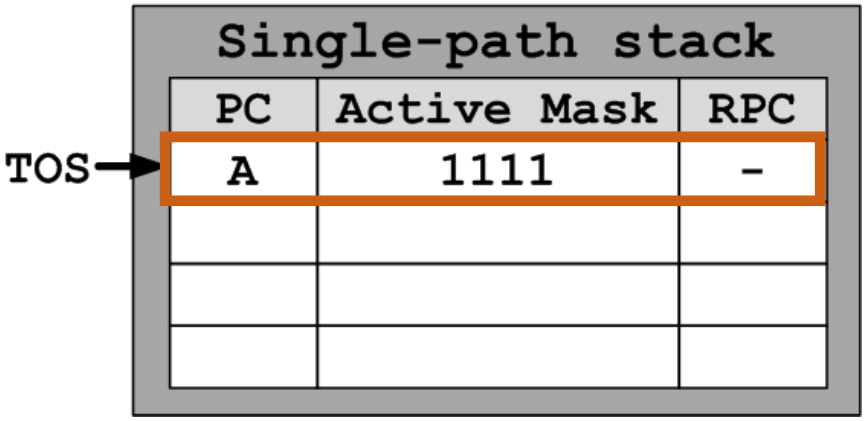
(b) Using the stack for control flow management

SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute *active*-threads at the ***top-of-stack*** (TOS)
  - **Active bitmasks** of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at *reconvergence point*
  - **Reconvergence PC** (RPC) point derived at compile-time
    - RPC: ***immediate post-dominator (PDOM)*** of the branching-point



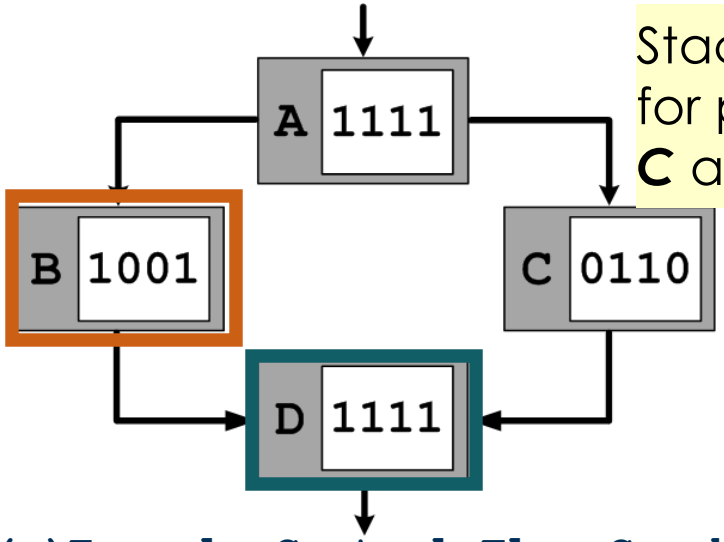
(a) Example Control Flow Graph
1: Active, 0: Inactive



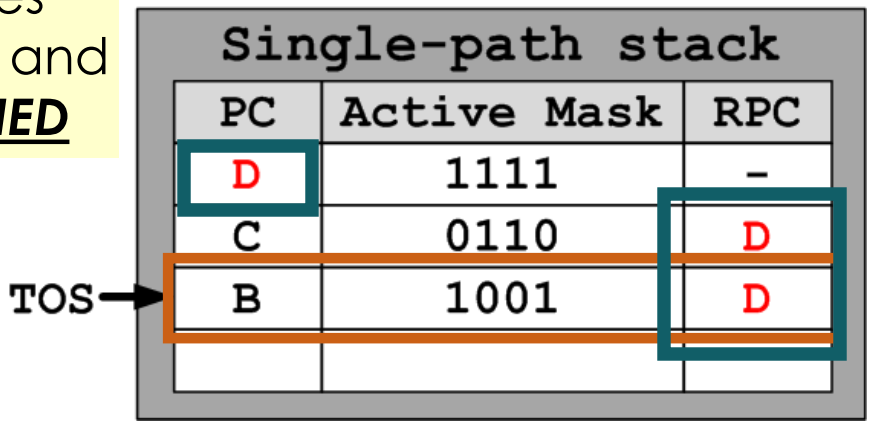
(b) Using the stack for control flow management

SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute *active*-threads at the ***top-of-stack*** (TOS)
  - **Active bitmasks** of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at *reconvergence point*
  - **Reconvergence PC** (RPC) point derived at compile-time
    - RPC: ***immediate post-dominator (PDOM)*** of the branching-point



Stack entries for paths ***B*** and ***C*** are ***PUSHED***

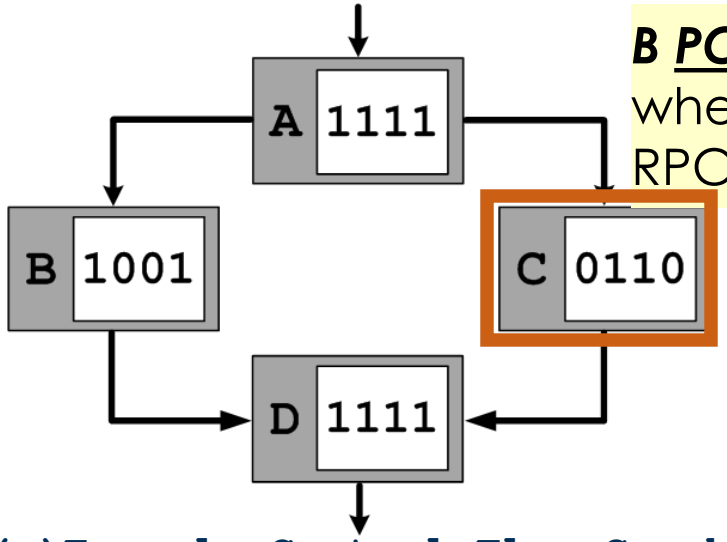


(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management

SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute *active*-threads at the ***top-of-stack*** (TOS)
  - **Active bitmasks** of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at *reconvergence point*
  - **Reconvergence PC** (RPC) point derived at compile-time
    - RPC: ***immediate post-dominator (PDOM)*** of the branching-point



B POPPED when reaching RPC (**D**)

Single-path stack

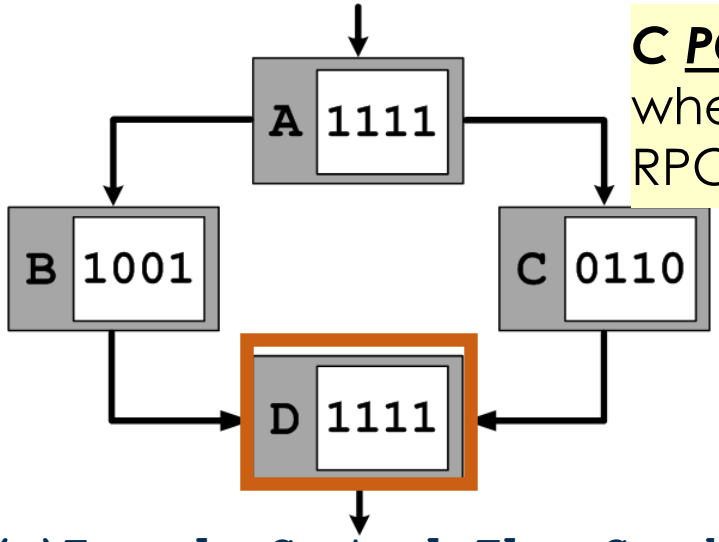
	PC	Active Mask	RPC
	D	1111	-
TOS →	C	0110	D

(a) Example Control Flow Graph
1: Active, 0: Inactive

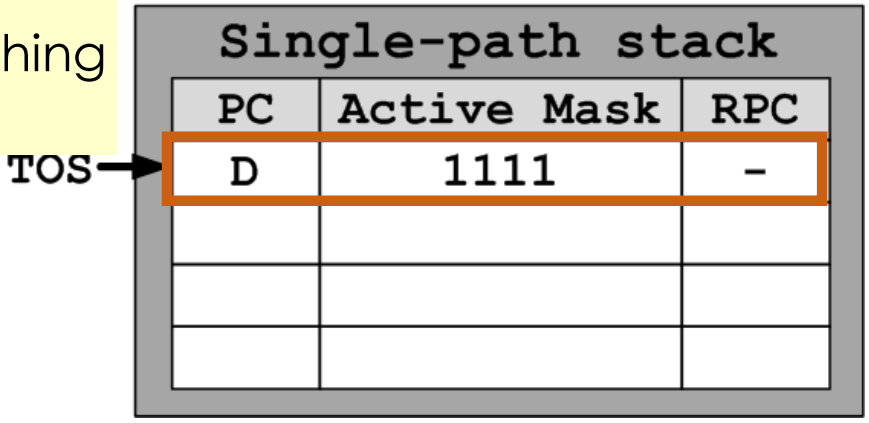
(b) Using the stack for control flow management

SIMT stack-based reconvergence model

- Current GPUs use per-warp HW stack to manage control flow
  - Always execute *active*-threads at the ***top-of-stack*** (TOS)
  - **Active bitmasks** of taken/not-taken path *dynamically* derived
  - Reconverge taken/not-taken path at *reconvergence point*
  - **Reconvergence PC** (RPC) point derived at compile-time
    - RPC: ***immediate post-dominator (PDOM)*** of the branching-point



C POPPED when reaching RPC (**D**)



(a) Example Control Flow Graph
1: Active, 0: Inactive

(b) Using the stack for control flow management

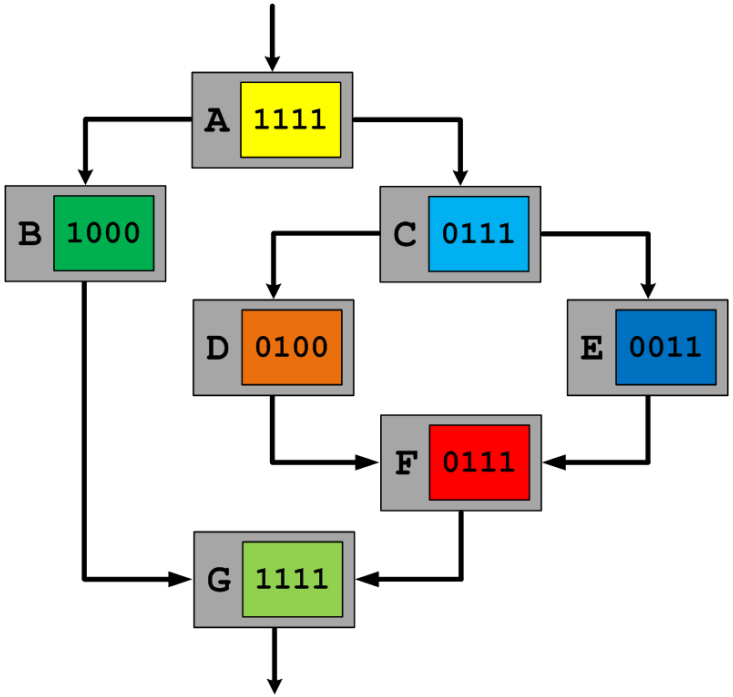
Is this dynamic adaptive HW?

- Yes!
  - Tuned for regularity
  - Supports irregular
- But irregular control often performs poorly
  - Only fraction of lanes is active
  - Reduced effective TLP
    - Paths serialized and each has fraction of active threads
  - Reduced performance and efficiency
    - SIMD units under utilized

Exploiting irregular control to improve parallelism

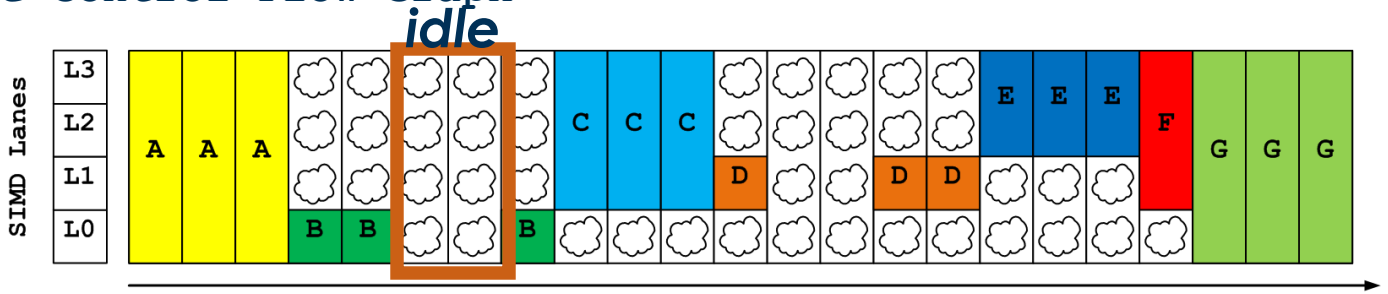
DUAL PATH EXECUTION

Serialization of execution paths



- Execution can transition to the next path *only* when ***all*** the instructions in the ***current*** path (basic-block) are executed.
  - CFG is managed with STACK!
- Misses opportunities to schedule from *other* ***concurrent*** paths
  - e.g., paths B/C

(a) Example Control Flow Graph



(b) Execution flow using baseline mechanism.

idle cycles: phases where the scheduler is short of warps to schedule (i.e. cache miss, long-latency ops)

Dual-Path Execution Model (DPE)

- Increase thread-level parallelism (or ***path-parallelism***)
  - Without sacrificing SIMD lane utilization
- Maintains the *simplicity* of the baseline stack model
- Minimal implementation overhead
  - Purely a microarchitectural solution
  - No additional compiler support needed

Dual-Path Execution Model (DPE)

- Increase thread-level parallelism (or ***path-parallelism***)
  - Without sacrificing SIMD lane utilization
  - Previous work ***compromises*** SIMD efficiency for enhanced TLP
- Maintains the *simplicity* of the baseline stack model
- Minimal implementation overhead
  - Purely a microarchitectural solution
  - No additional compiler support needed

- ***Meng et al., "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence", ISCA-2010***

DPE components: dual-path stack

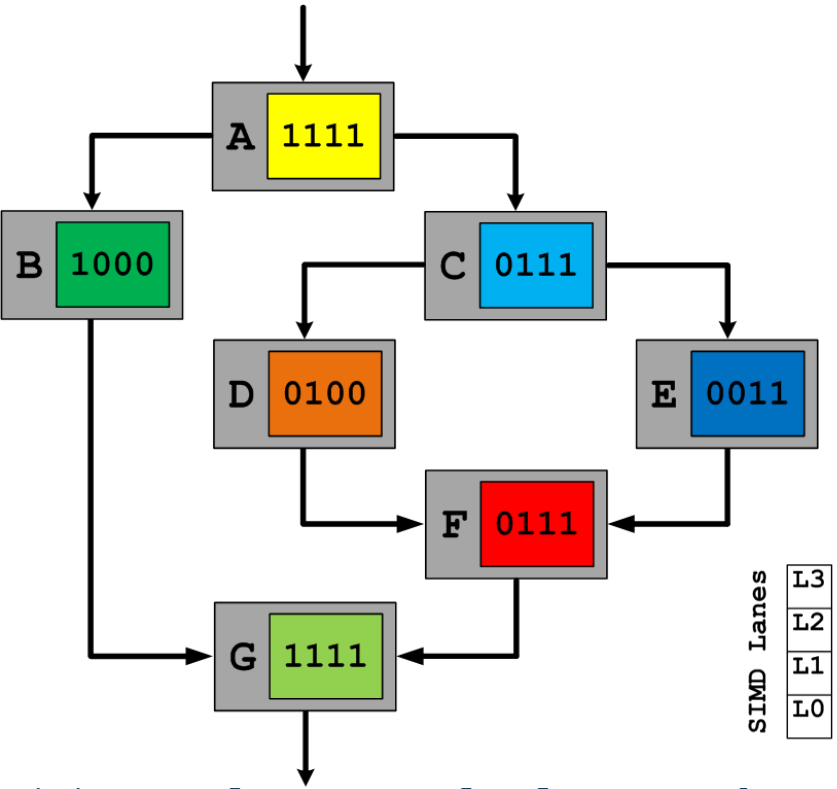
- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

RPC value is **shared** by both Left & Right paths (always at *PDOM*)

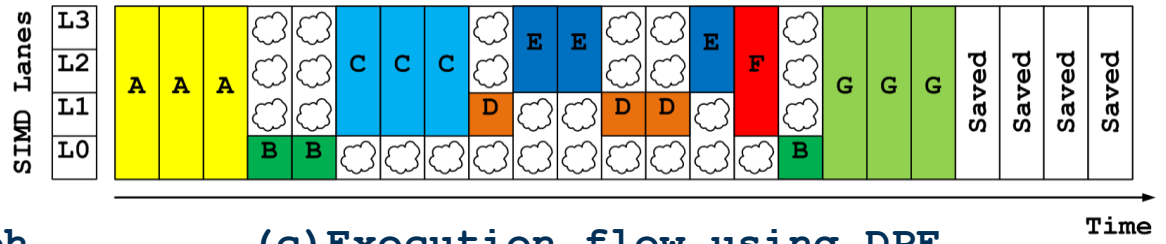
Dual-path stack

	PC_L	$Mask_L$	PC_R	$Mask_R$	RPC
TOS →	A	1111	-	-	-

(b) Using the DPE stack for control flow management



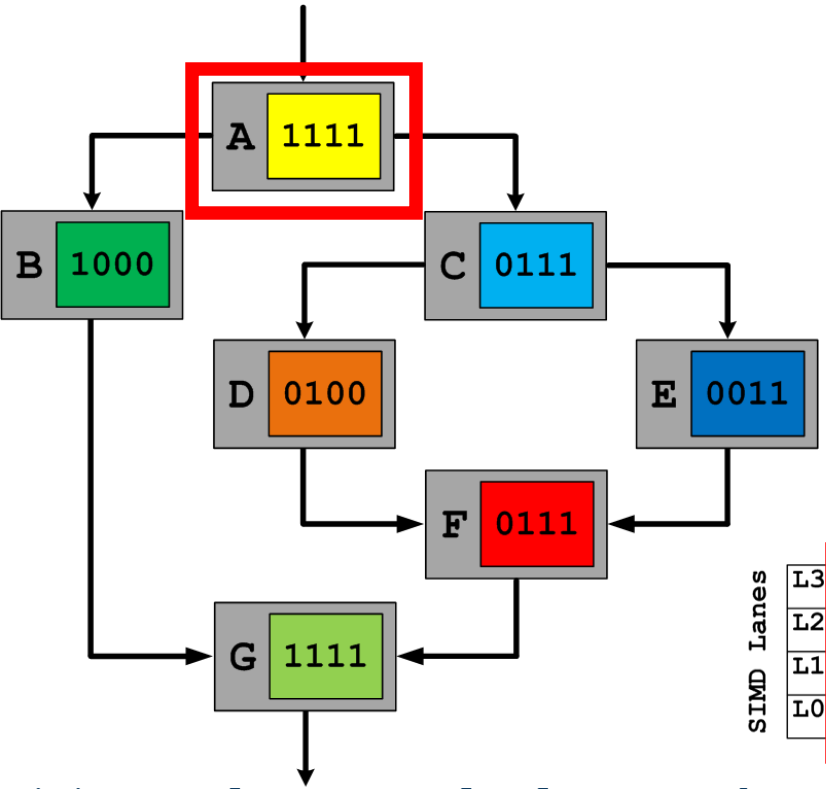
(a) Example Control Flow Graph



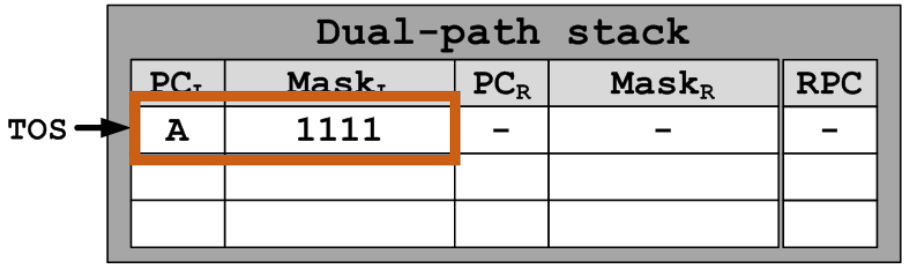
(c) Execution flow using DPE.

DPE components: dual-path stack

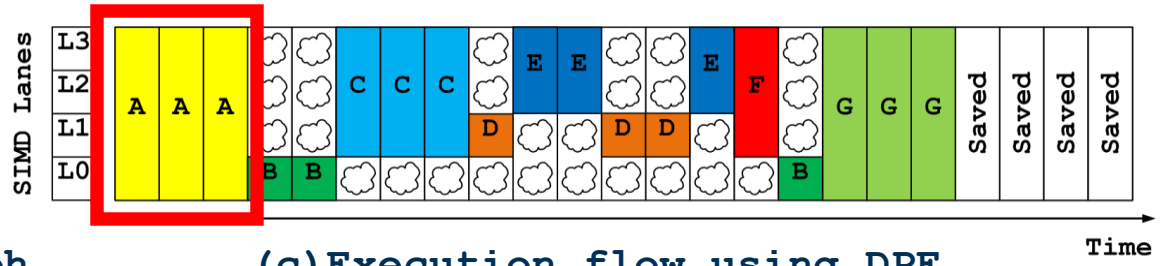
- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE



(a) Example Control Flow Graph



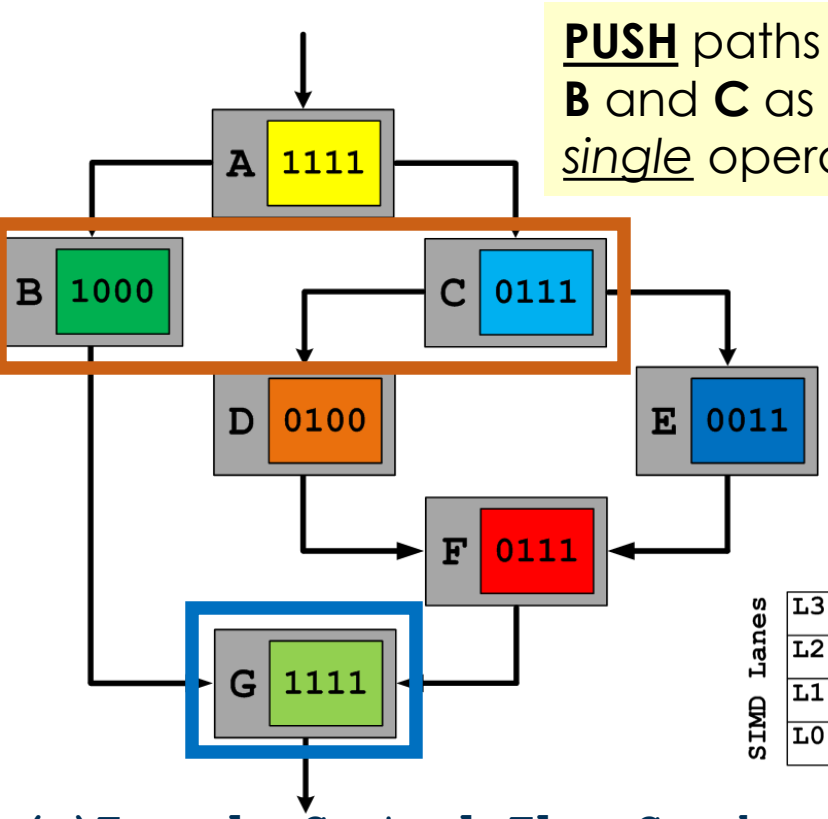
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

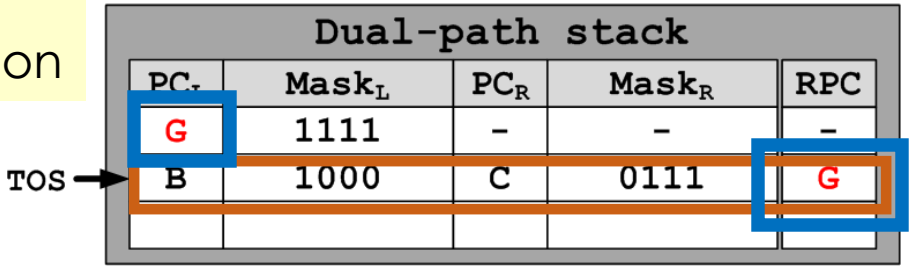
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

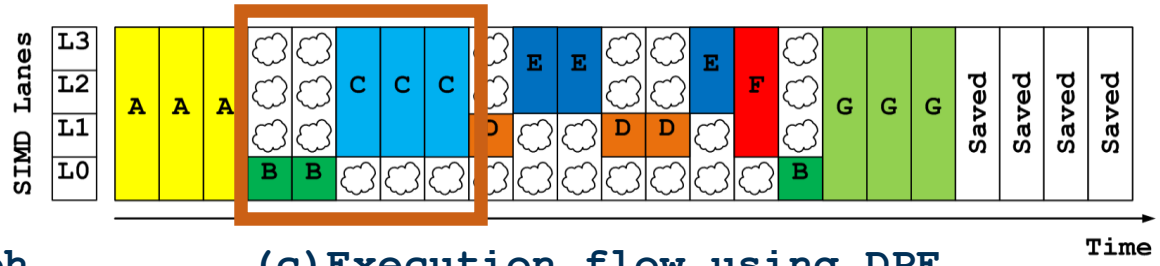


(a) Example Control Flow Graph

PUSH paths **B** and **C** as a *single* operation



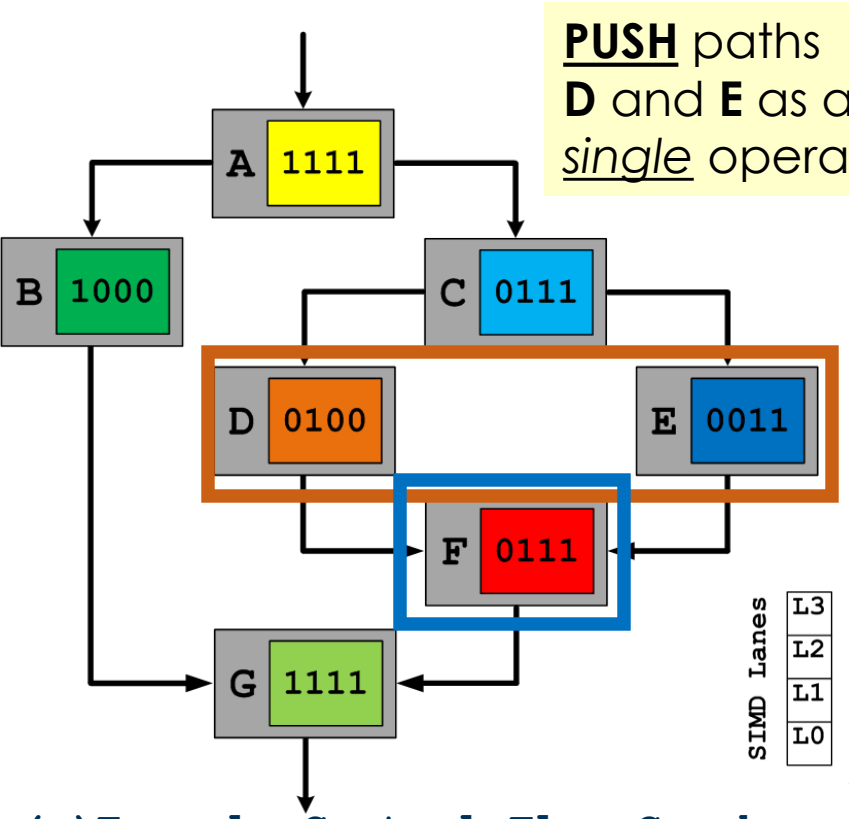
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

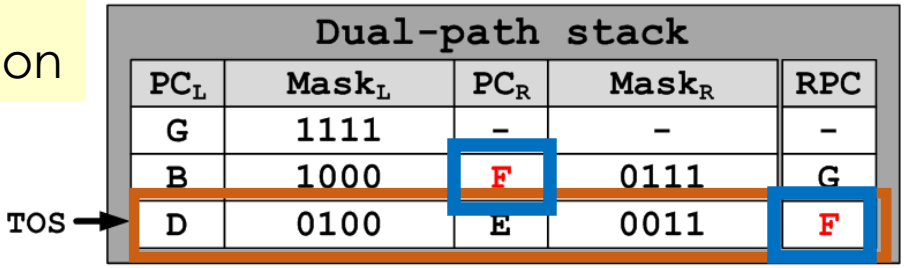
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

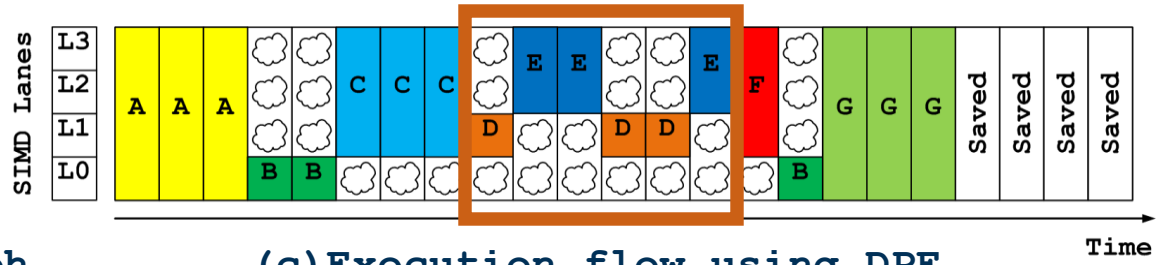


(a) Example Control Flow Graph

PUSH paths **D** and **E** as a *single* operation



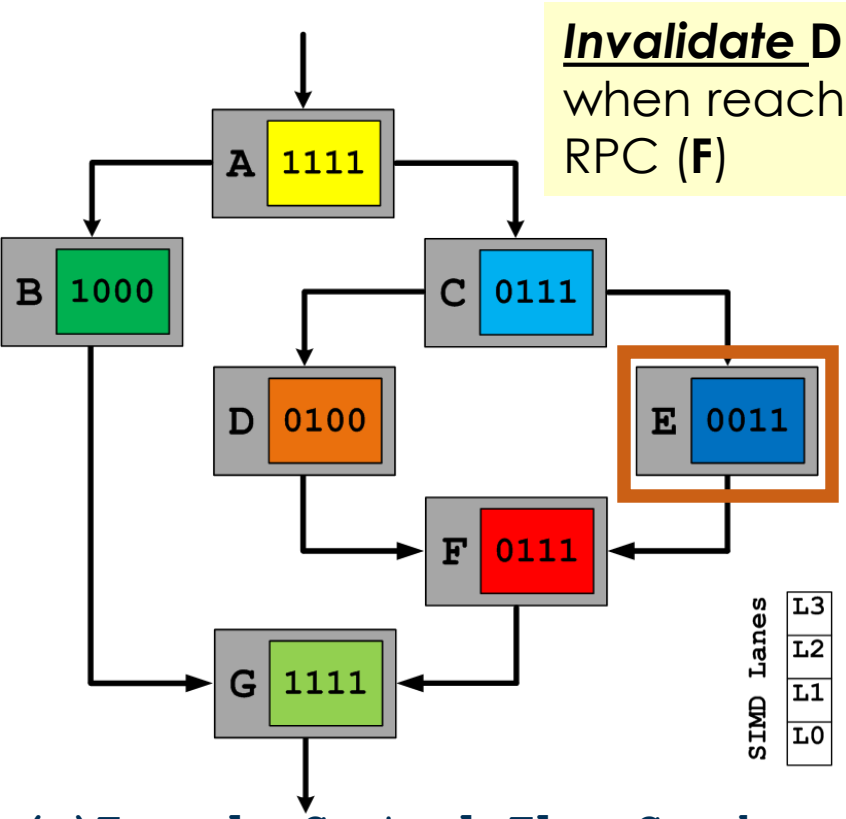
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

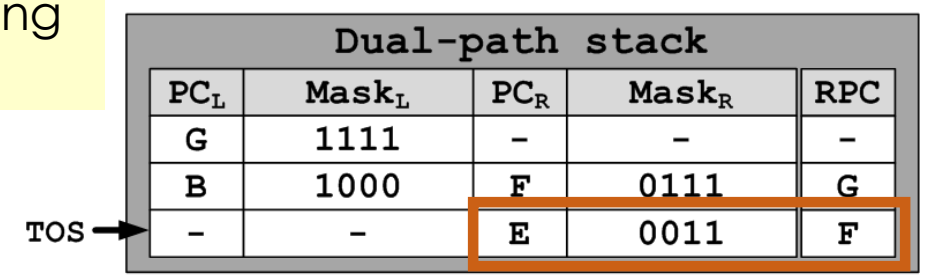
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

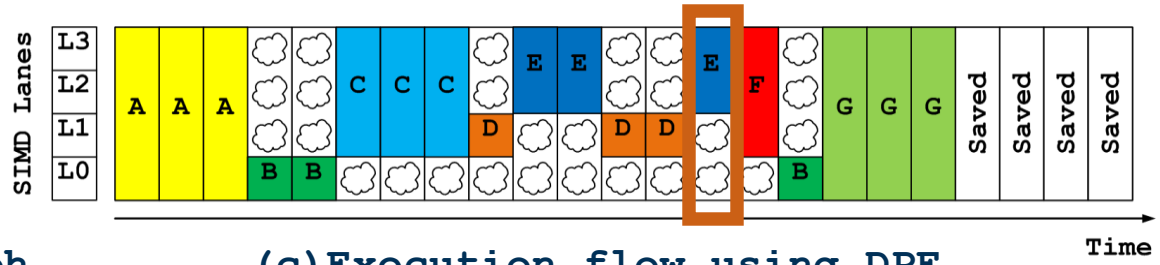


(a) Example Control Flow Graph

Invalidate **D** when reaching RPC (**F**)



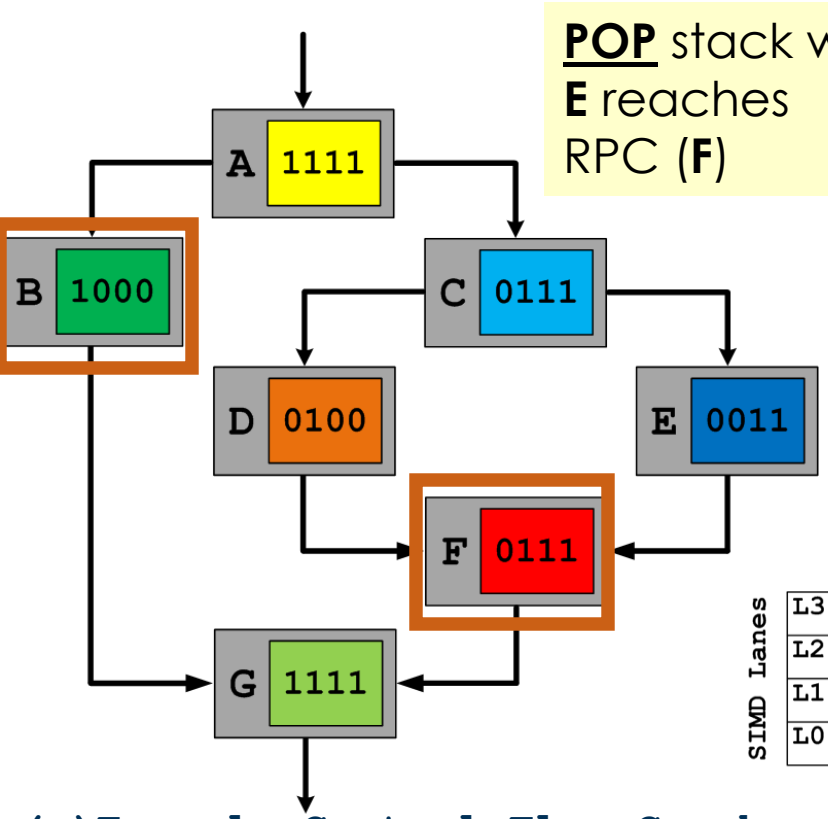
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

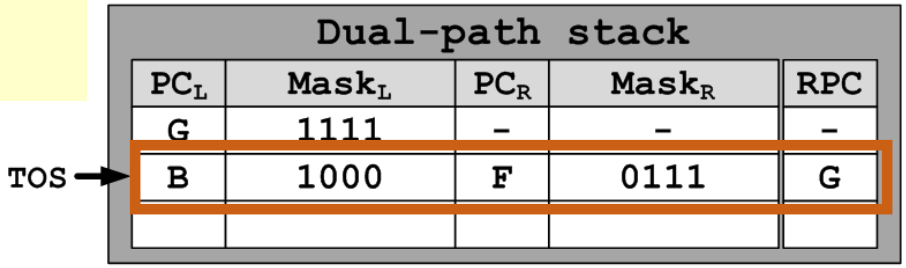
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

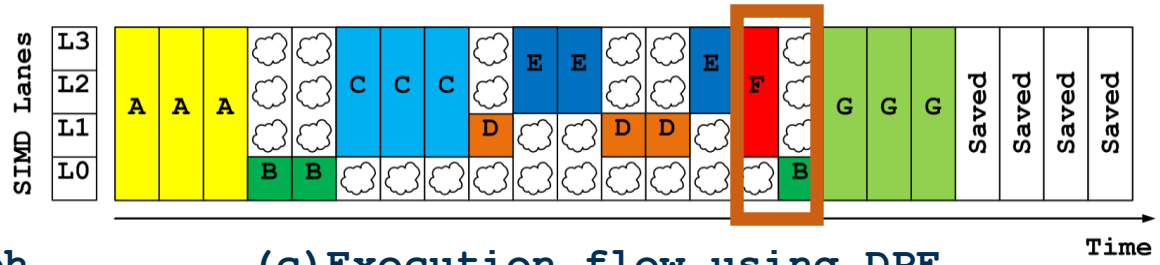


(a) Example Control Flow Graph

POP stack when **E** reaches RPC (**F**)



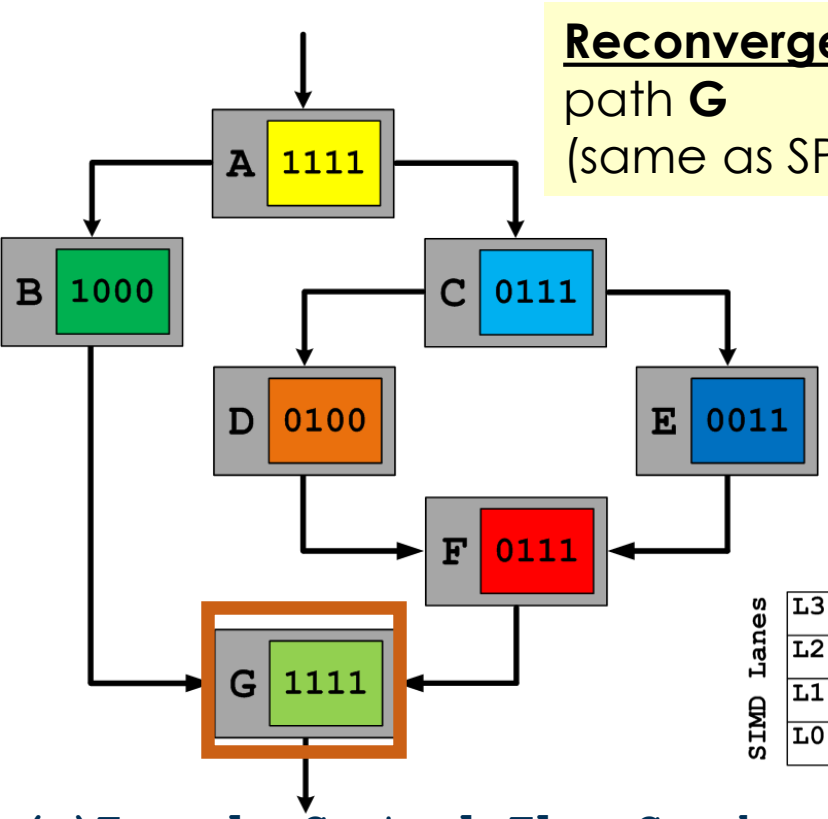
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

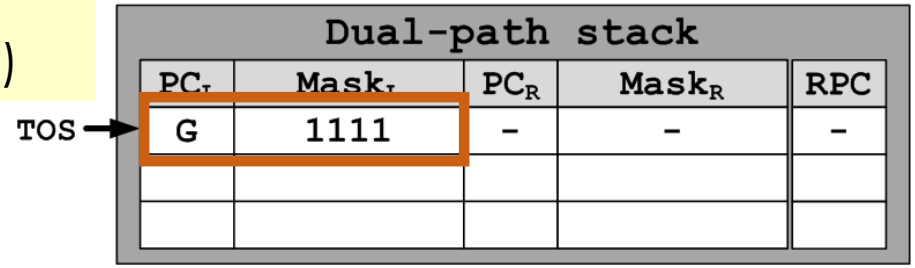
DPE components: dual-path stack

- DPE stack microarchitecture
  - Each entry accommodates ***both*** paths of a branching point
  - Single dual-path entry instead of two separate entries in SPE

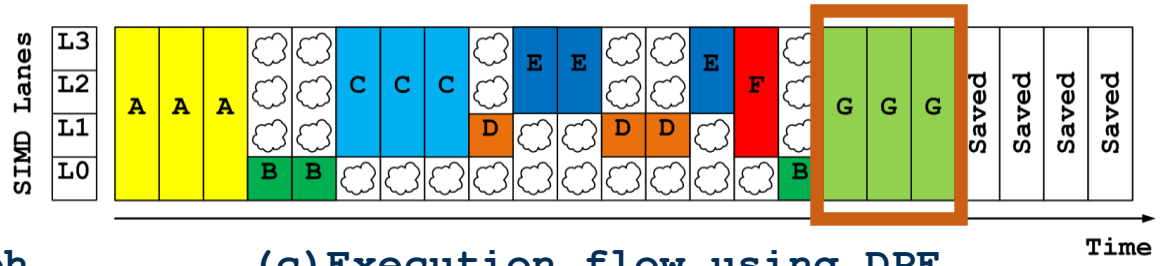


(a) Example Control Flow Graph

Reconverge at path **G** (same as SPE)



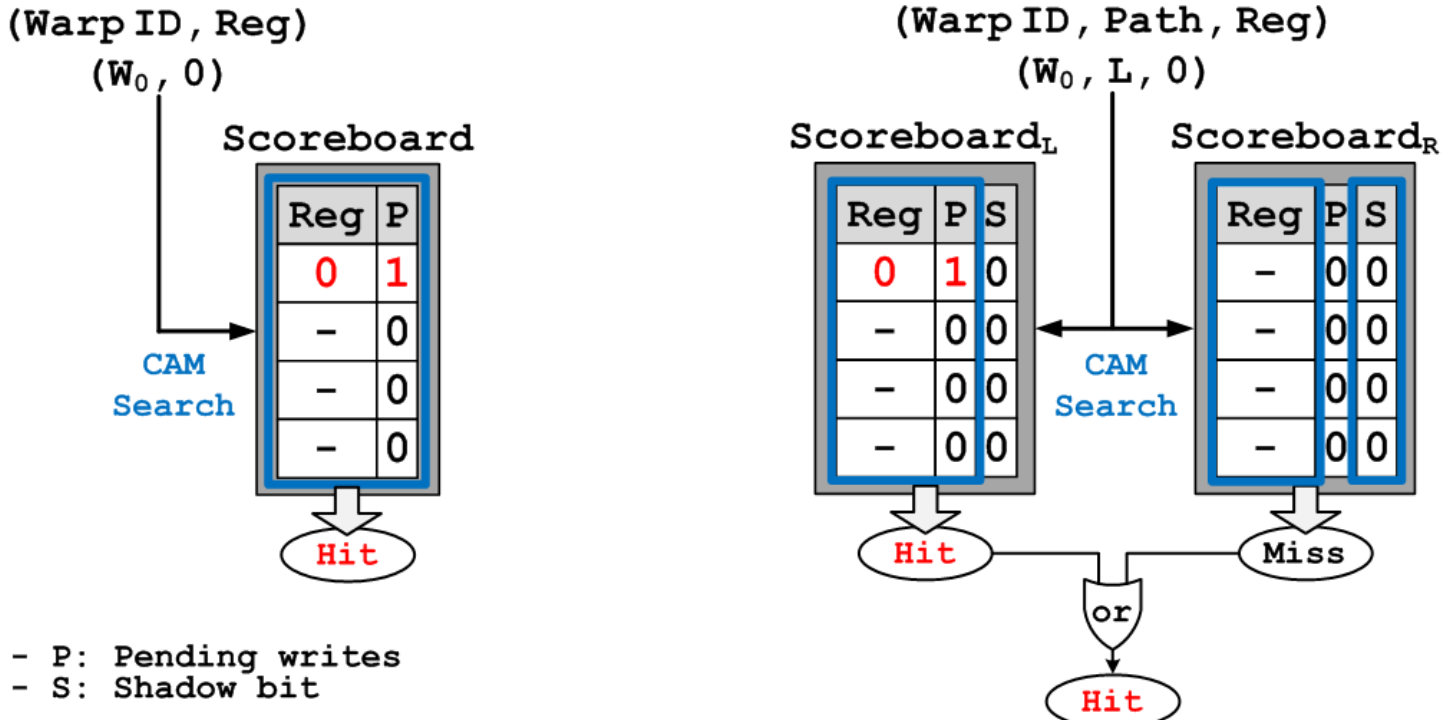
(b) Using the DPE stack for control flow management



(c) Execution flow using DPE.

DPE components: scoreboard

- Current GPUs execute ***intra***-warp threads back-to-back
- DPE complications because T/NT paths share registers
  - Separate Left/Right scoreboards
  - Shadow-bits for pre-/post-divergence dependencies



DPE components: warp scheduler

- Scheduler enhanced to cover ***both*** T/NT paths
  - For maximal benefits of DPE, scheduler overhead is doubled
  - *'Constrained'* scheduler
    - Warp-scheduler is fed with only a single path at the TOS
    - Path to be sent to the scheduler is rotated when a *long-latency operation* is executed
    - Benefits decrease from **14.9%** to **11.7%**

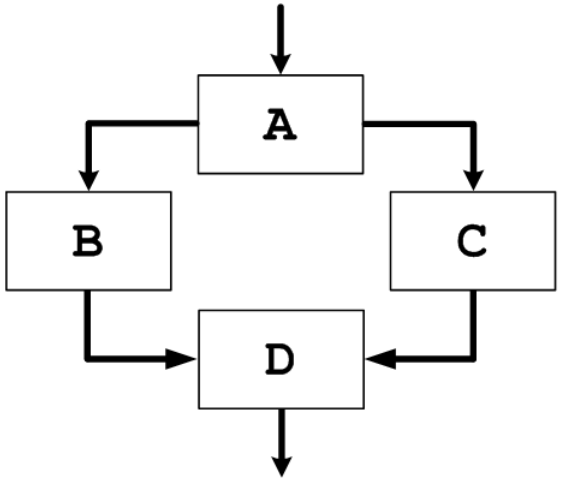
Simulation environment

- GPGPU-Sim
  - Cycle-based performance simulator of a GPGPU
  - 15 shader cores (Streaming Multiprocessors, SMs)
  - 1536 threads per core, 32K registers per core
  - Cache: 16kB L1, 768kB Unified L2
  - Shared-memory: 48KB
  - Memory Controller : FR-FCFS
    - 29.6GB/s per channel, 6 channels overall
- Applications
  - Chosen from CUDA-SDK, Rodinia, Parboil, Cuda Zone, etc
  - Will focus on apps exhibiting distinct behavior across different schemes

Average path parallelism

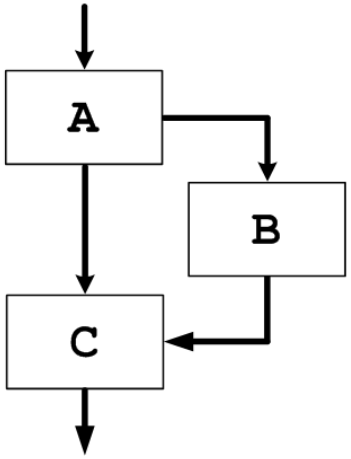
- Num of concurrent paths exposed to the scheduler
  - SPE: always '1'
  - DPE: '2' if both L/R path available, and '1' otherwise

< Corresponding control flow graph >



- Interleavable branch
  - Both "**if**"/"**else**" part active
  - DPE scheduler sees '2' paths

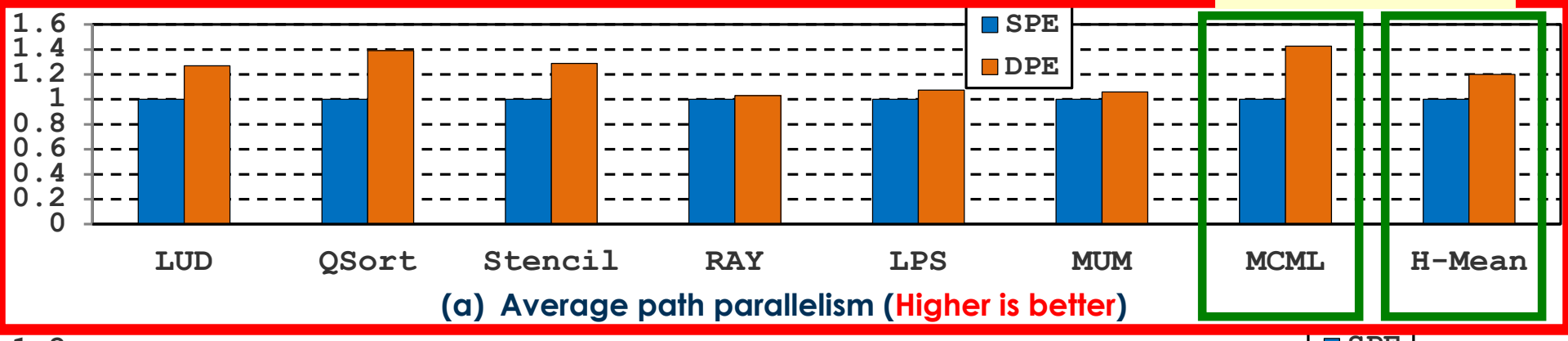
< Corresponding control flow graph >



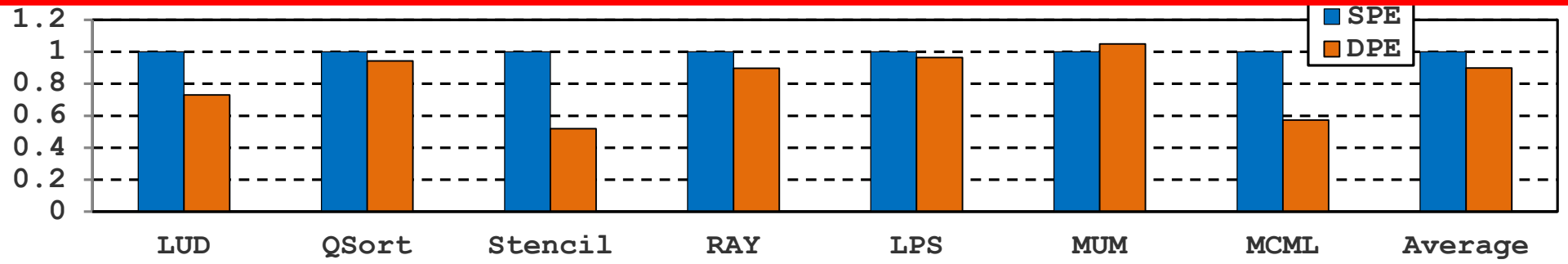
- Non-interleavable branch
  - ***No*** '**else**' part
  - DPE scheduler sees '1' path

Avg. path parallelism / idle cycles / speedup

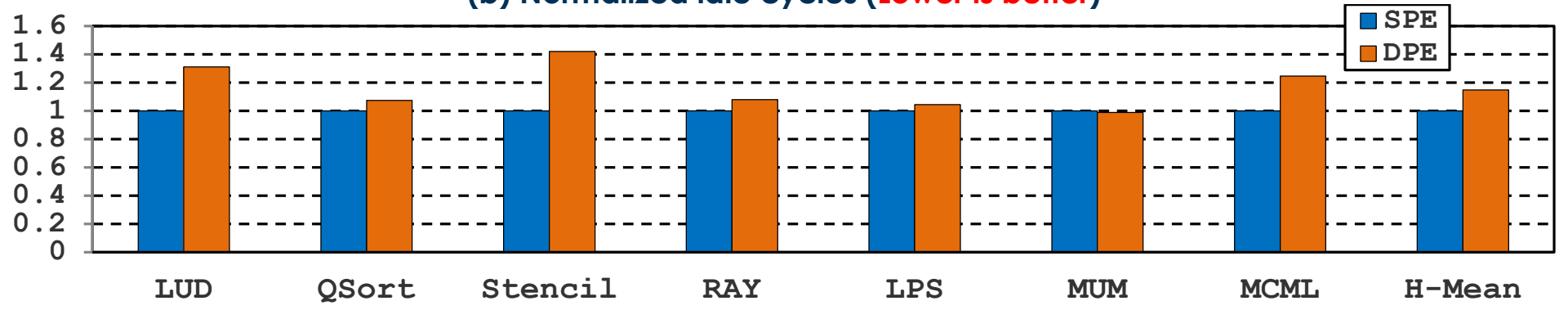
Max: 42.5% ↑
Avg: 20.1% ↑



(a) Average path parallelism (Higher is better)

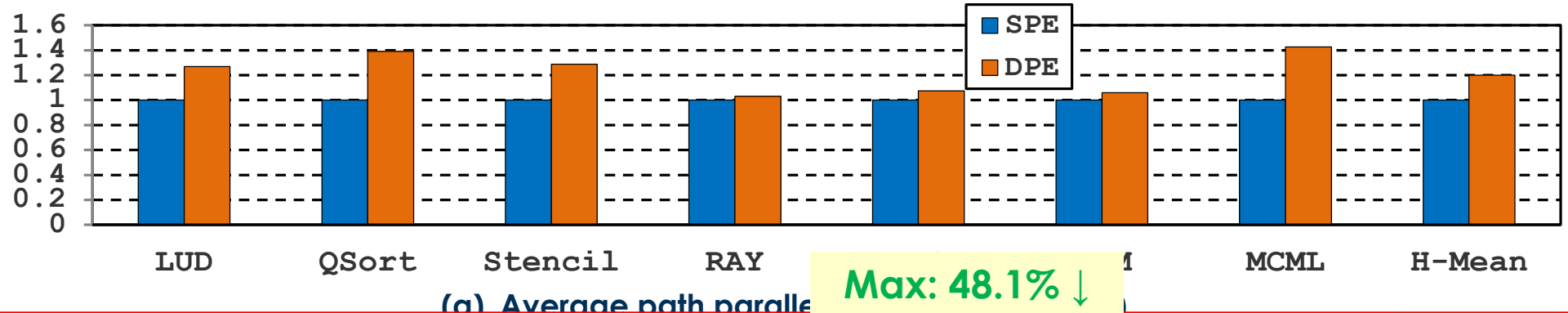


(b) Normalized idle cycles (Lower is better)



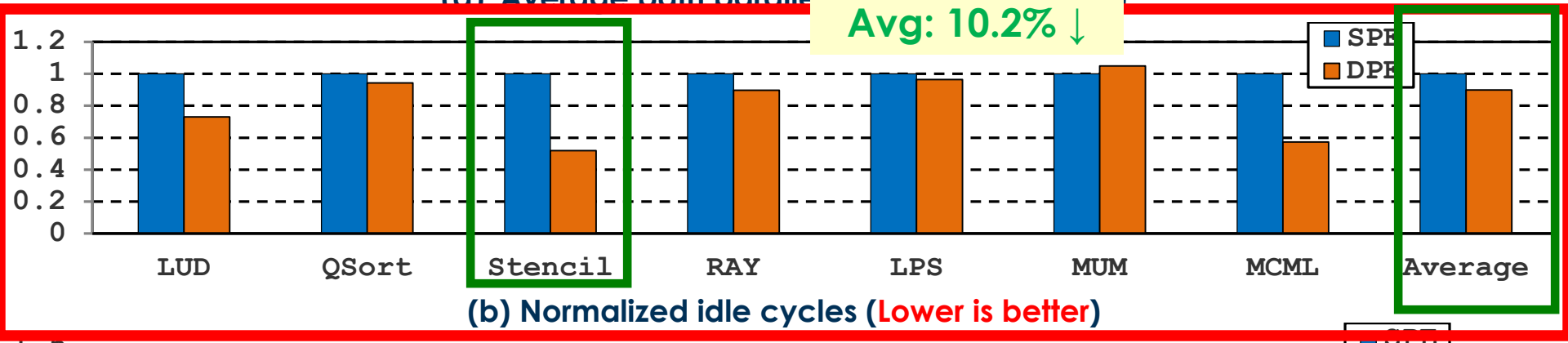
(c) Speedup (Higher is better)

Avg. path parallelism / idle cycles / speedup

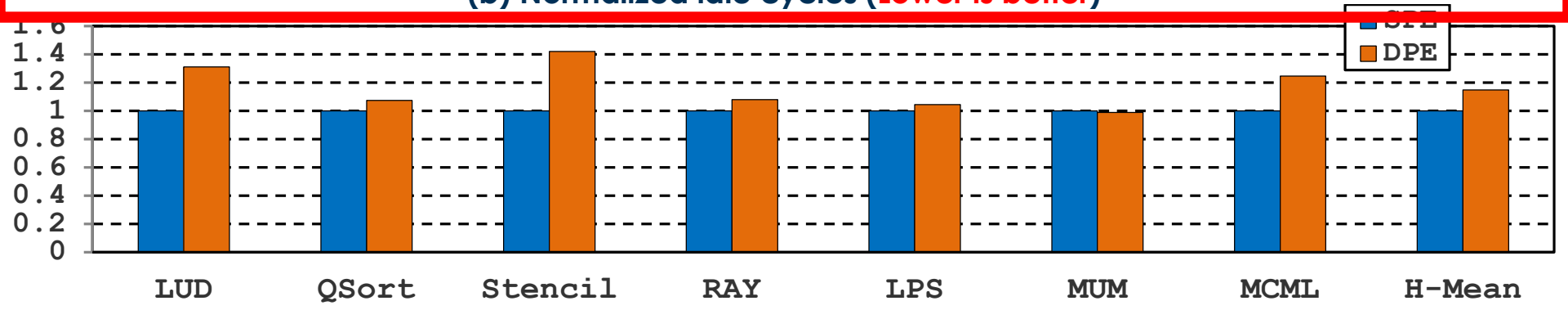


(a) Average path parall

Max: 48.1% ↓
Avg: 10.2% ↓

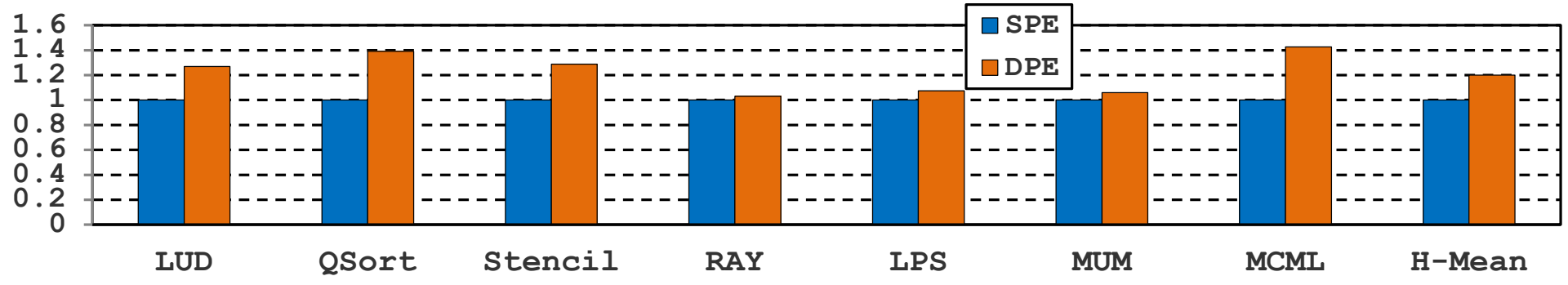


(b) Normalized idle cycles (Lower is better)

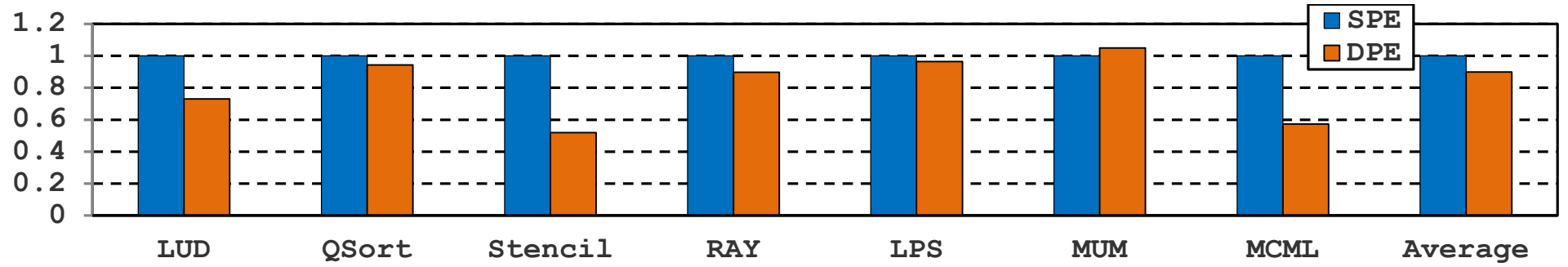


(c) Speedup (Higher is better)

Avg. path parallelism / idle cycles / speedup

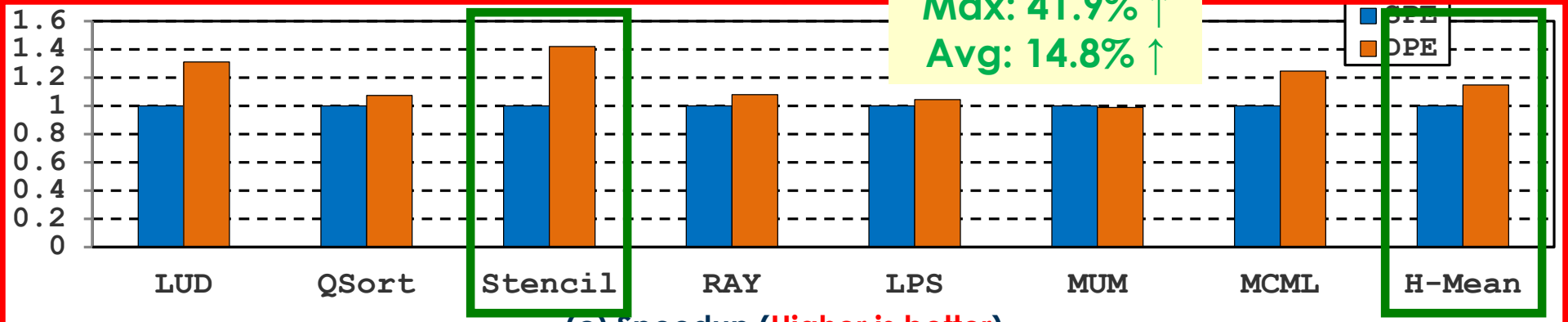


(a) Average path parallelism (Higher is better)



(b) Normalized idle cycles (L

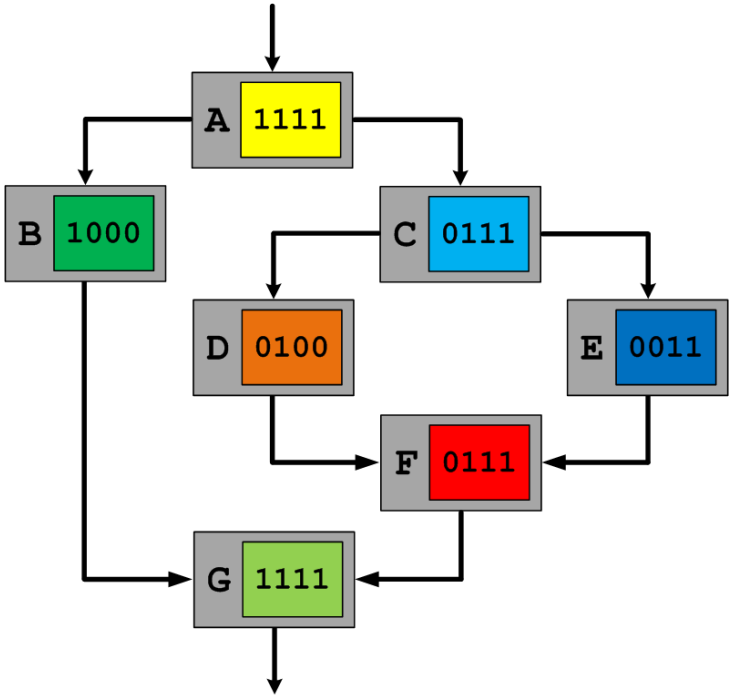
Max: 41.9% ↑
Avg: 14.8% ↑



(c) Speedup (Higher is better)

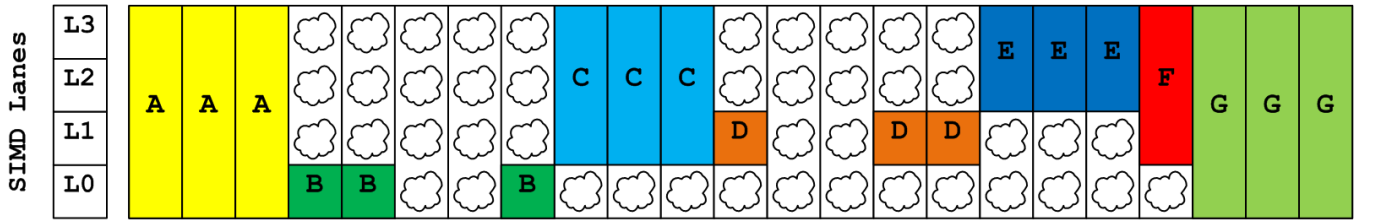
IMPROVING SIMD EFFICIENCY

Underutilization of SIMD units



- Number of active threads in a warp decreases every time control diverges
- Theoretically, only a single SIMD lane might be active at some point, for highly divergent apps.
  - Nested divergent branches

(a) Example Control Flow Graph



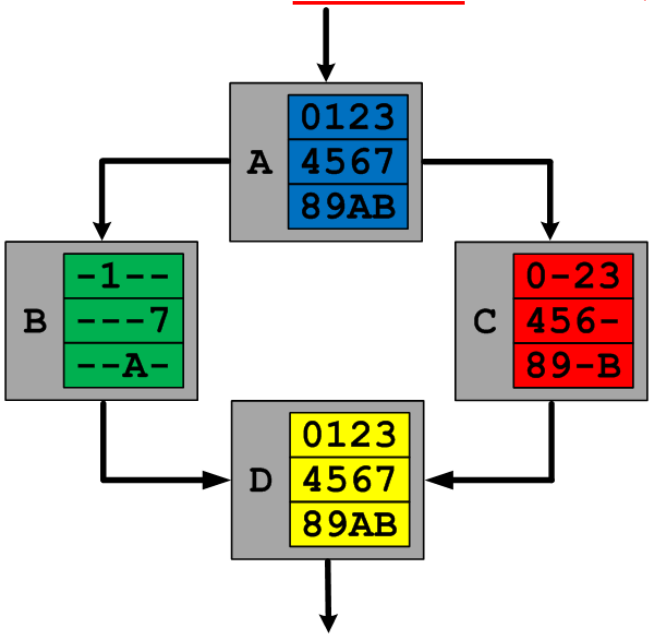
(b) Execution flow using baseline mechanism.

: Threads (lanes) masked out due to control divergence, remaining idle.

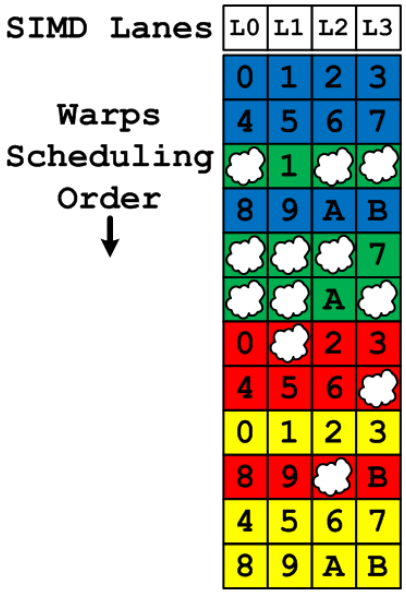
Thread-block compaction (TBC) [Fung'11]

- Dynamically ***compact*** warps within a ***thread-block***
  - ***Synchronize*** all warps at branches and reconvergence points
  - Threads with ***different*** home SIMD lanes compacted together

: [0,1… A,B] refers to *active* thread-IDs executing in that basic block
: [-] refers to *inactive* threads, masked out from execution



(a) Example control flow graph

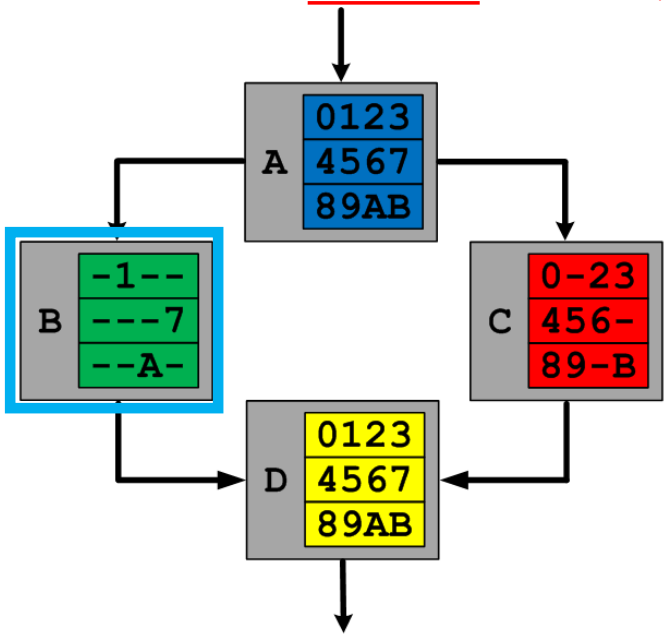


(b) Execution flow without compaction

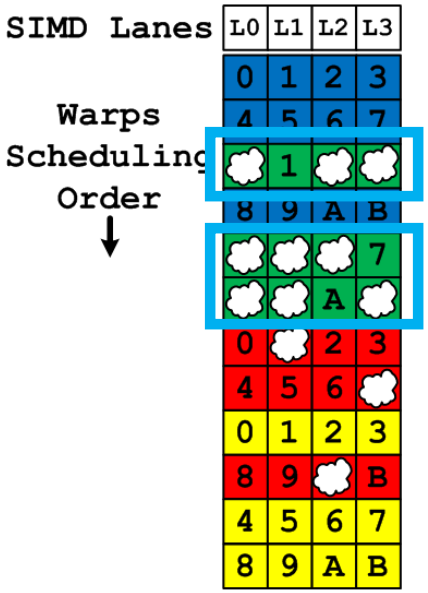
Thread-block compaction (TBC) [Fung'11]

- Dynamically ***compact*** warps within a ***thread-block***
  - *Synchronize* all warps at branches and reconvergence points
  - Threads with ***different*** home SIMD lanes compacted together

: [0,1⋯ A,B] refers to *active* thread-IDs executing in that basic block
: [-] refers to *inactive* threads, masked out from execution



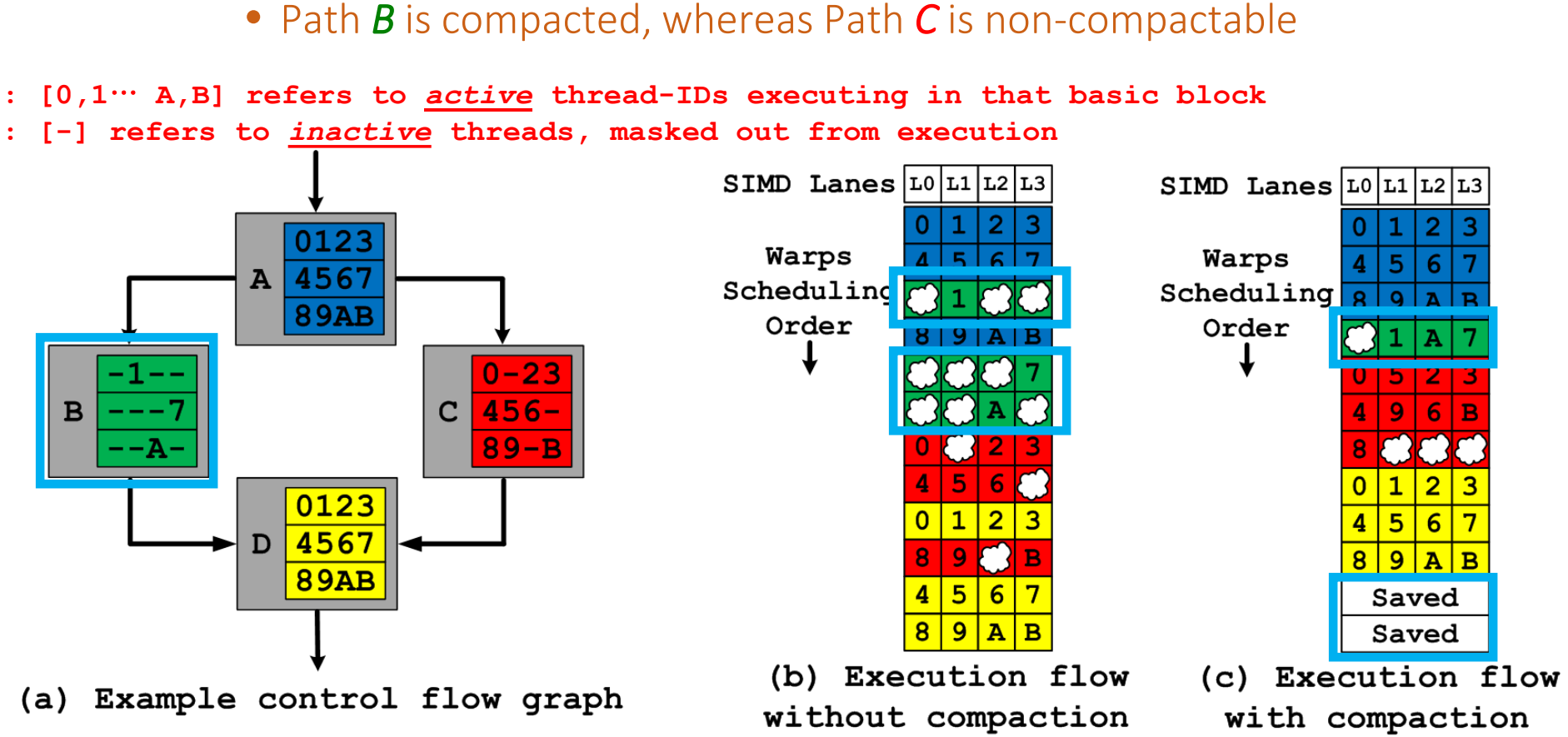
(a) Example control flow graph



(b) Execution flow without compaction

Thread-block compaction (TBC) [Fung'11]

- Dynamically ***compact*** warps within a ***thread-block***
  - *Synchronize* all warps at branches and reconvergence points
  - Threads with ***different*** home SIMD lanes compacted together
    - Path ***B*** is compacted, whereas Path ***C*** is non-compactable

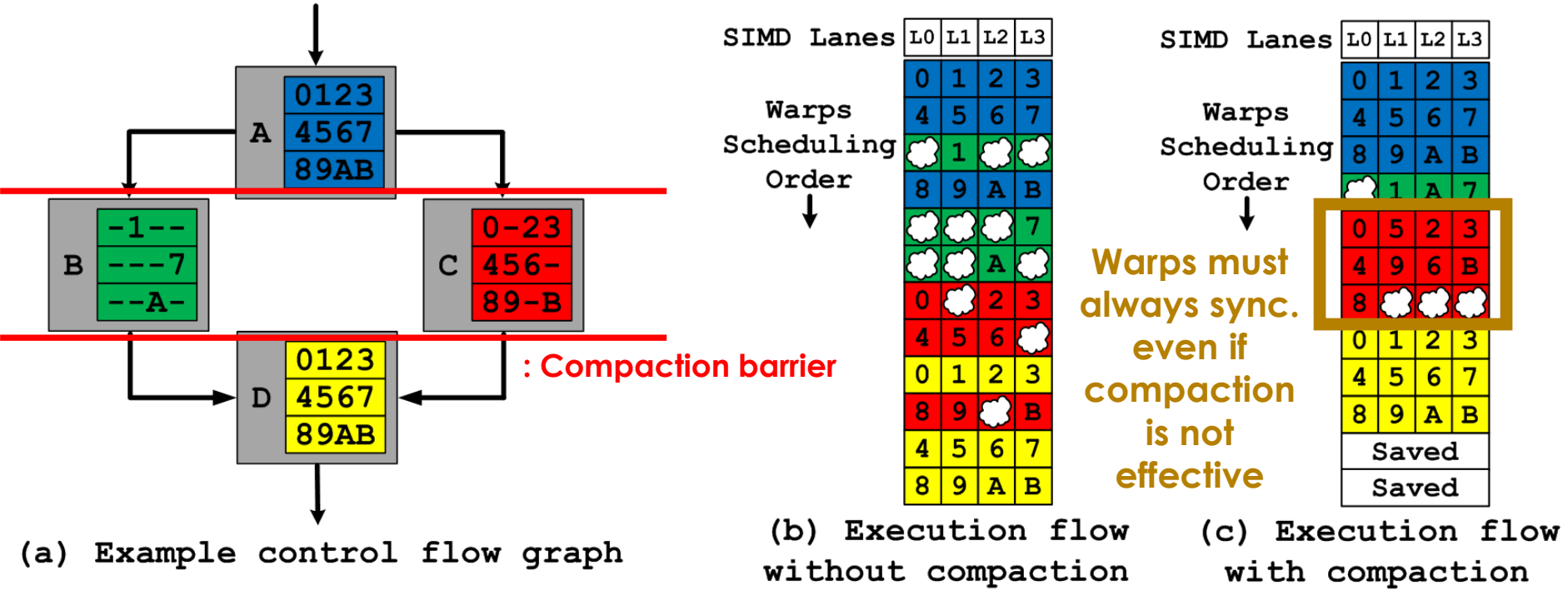


Is compaction dynamic adaptive HW?

- Yes!
  - Better support for diverged control flow
- But …
  - Regular(ized) control may suffer
  - Doesn’t work as is for many diverged branches

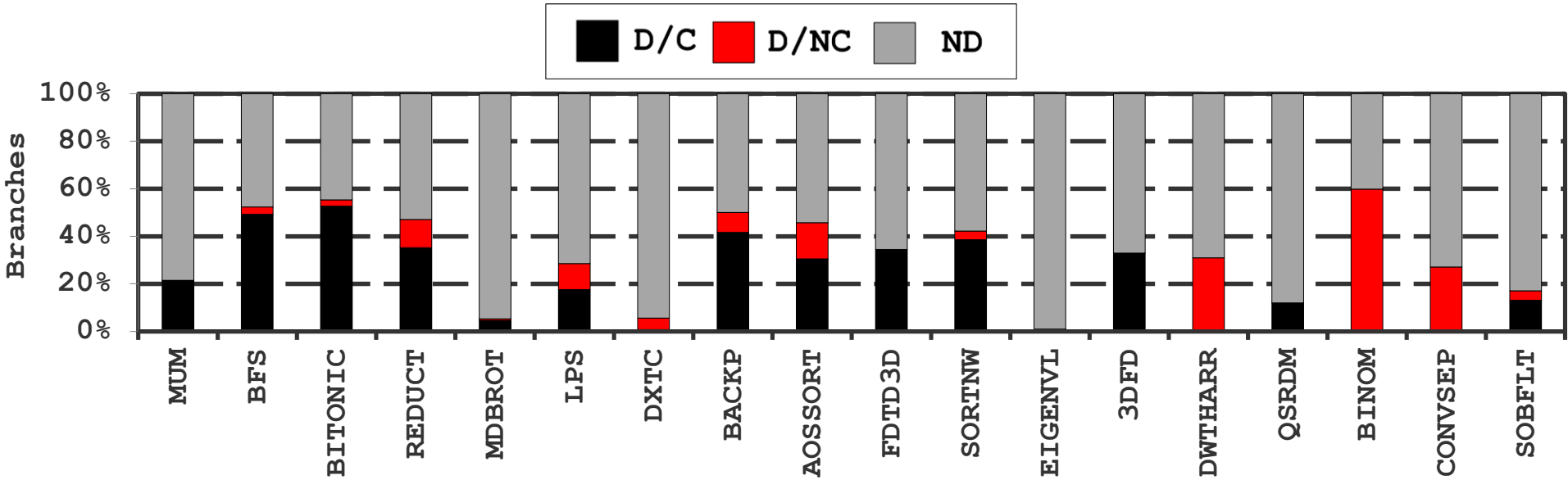
Excessive synchronization overhead

- Compaction is applied at ***all*** branching points
  - Effectively generates a HW-induced compaction-barrier
  - Warps arriving at the end of BB ***must*** wait for other warps in its CTA
  - Don't compact at ***unconditional*** branches [Narasiman'11]
    - TBC+ (TBC with no synchronization at unconditional branches)



Branch divergence & *ideal* compactability*

- *Conditional* branches, categorized as:
  - Divergent and *ideally* compactable (**D/C**)
  - Divergent but non-compactable *even with ideal* (**D/NC**)
  - Non-divergent (**ND**)



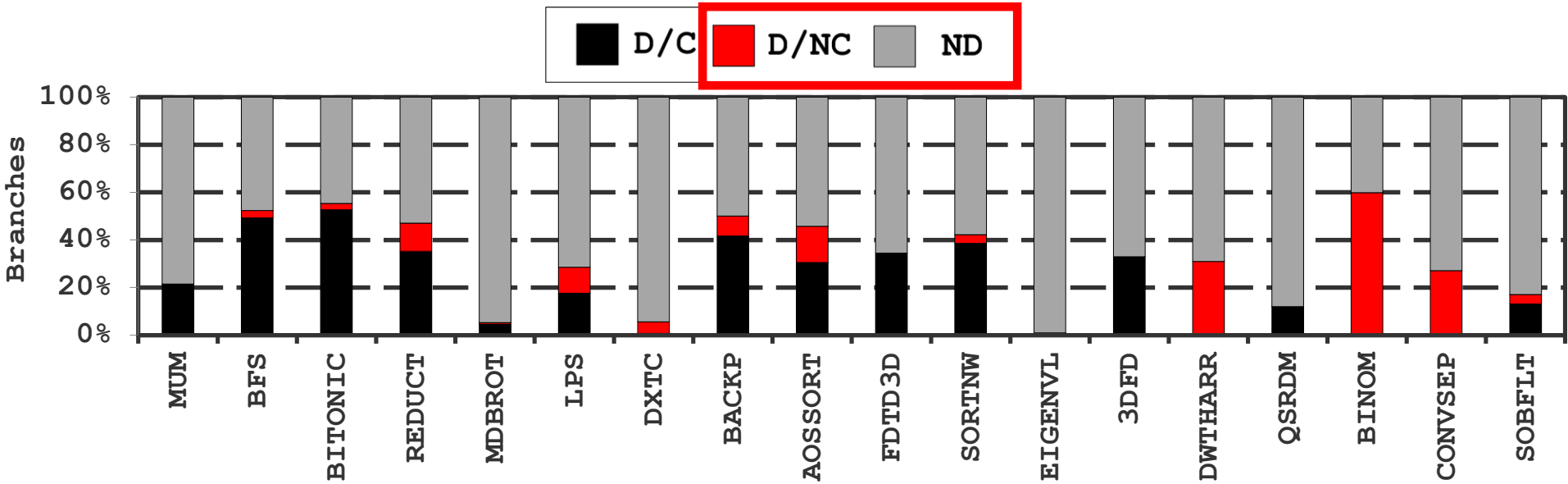
* Ideally compactable branches: branches that can be compacted if threads can switch its executing SIMD lanes

Branch divergence & *ideal* compactability*

- *Conditional* branches, c
  - Divergent and *ideally* c
  - Divergent but non-com
  - Non-divergent (**ND**)

Most conditional branches are ND or D/NC !

: *Why wait compaction ?*



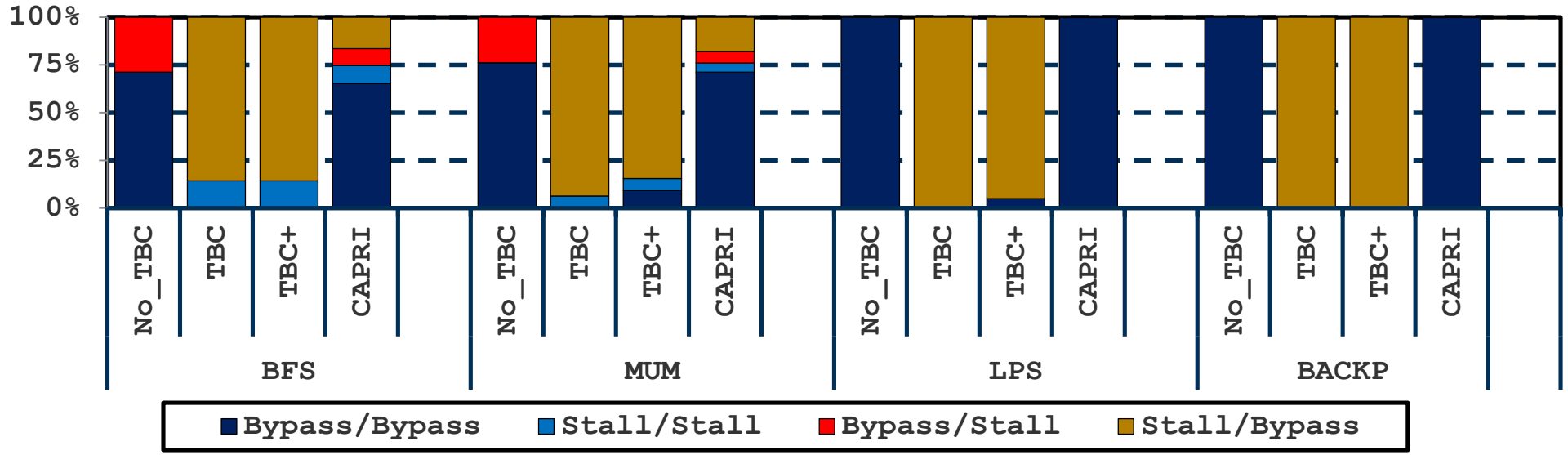
* Ideally compactable branches: branches that can be compacted if threads can switch its executing SIMD lanes

CAPRI: Compaction-Adequacy PRedIction

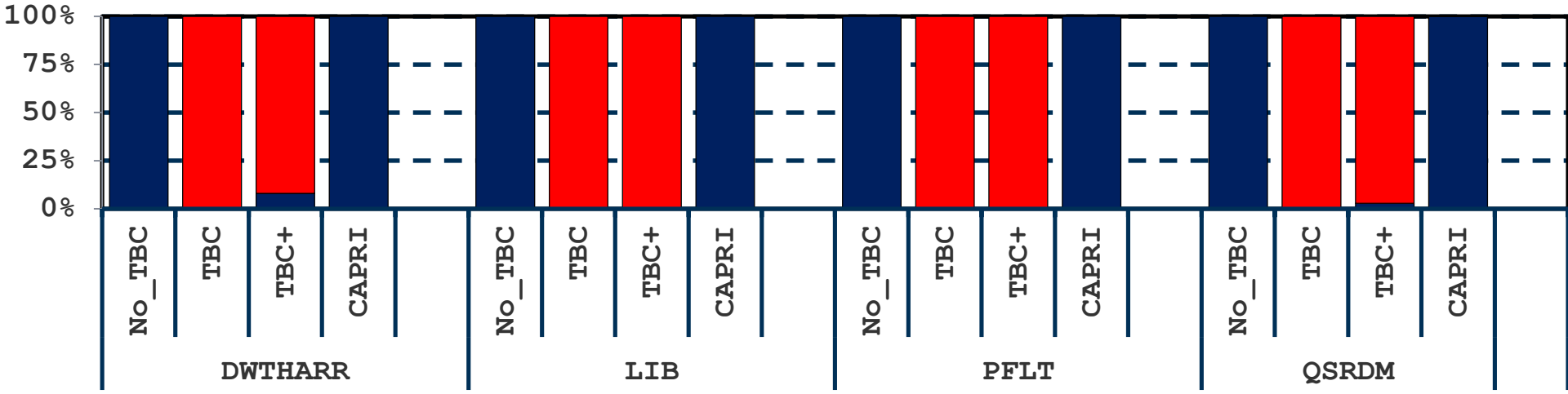
- Intuition
  - Not all branch points are likely to be compactable
  - *Speculate* whether compaction is worth it
    1) Activate compaction ***only when*** necessary
       - When past history says compaction was beneficial
    2) ***Bypass*** warps from compaction-barrier when inadequate
- Compaction-adequacy
  - Compaction is ***adequate*** when the number of executing warps was reduced at a particular branch
- Microarchitecture*
  - Per-core **prediction table** (32-entry, **TAG**: PC of branch)
  - Works like a simple single-level branch predictor

**** Rhu et al., "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures", ISCA-2012***

CAPRI accuracy

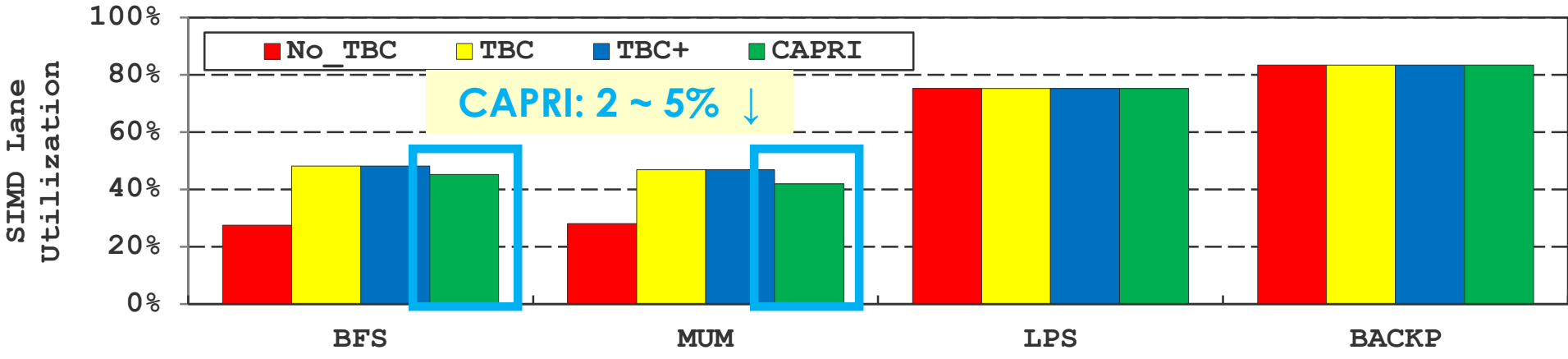


(a) Divergent Benchmarks



(b) Non-divergent Benchmarks

Average SIMD lane utilization*



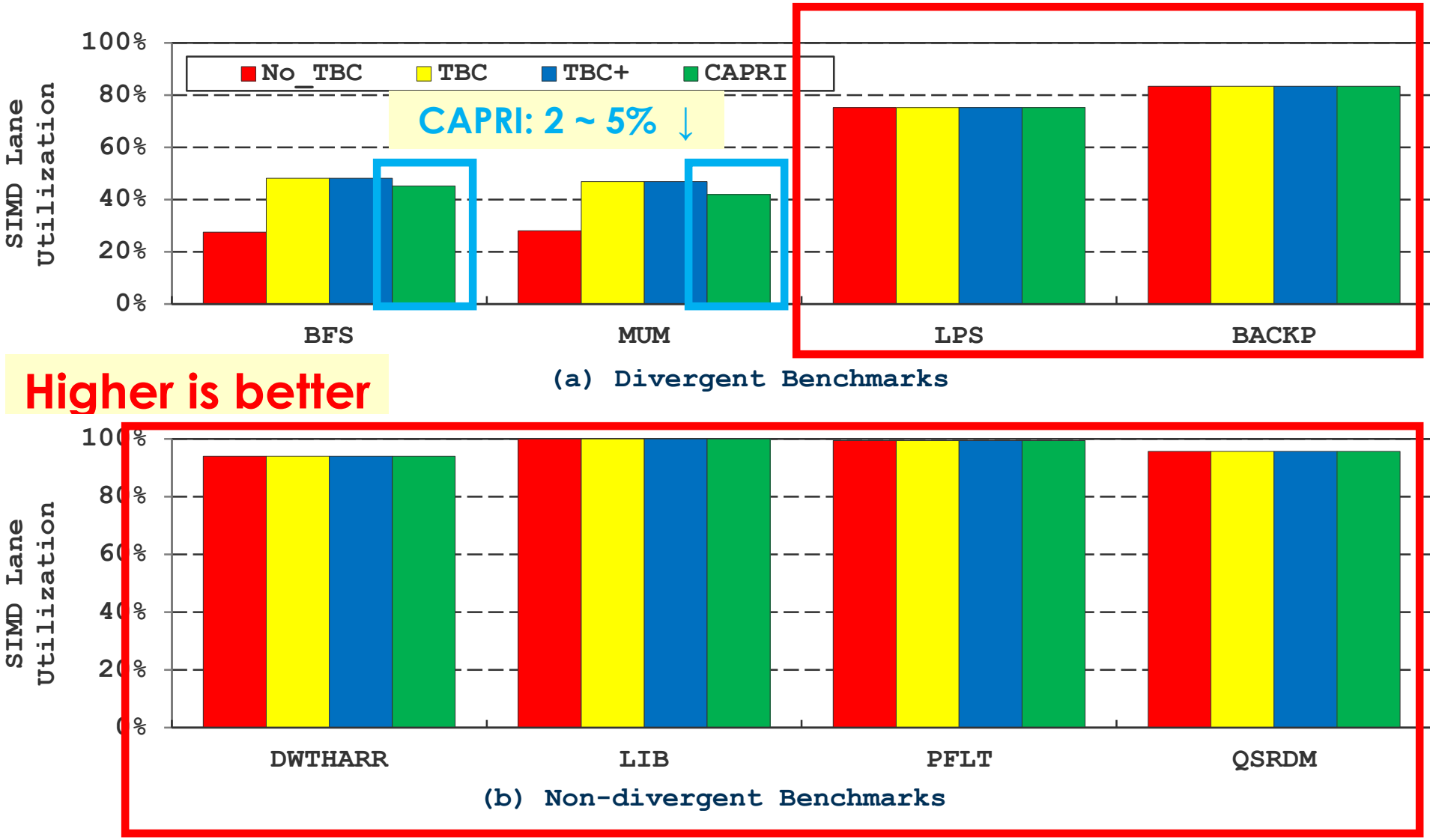
(a) Divergent Benchmarks

Higher is better

(b) Non-divergent Benchmarks

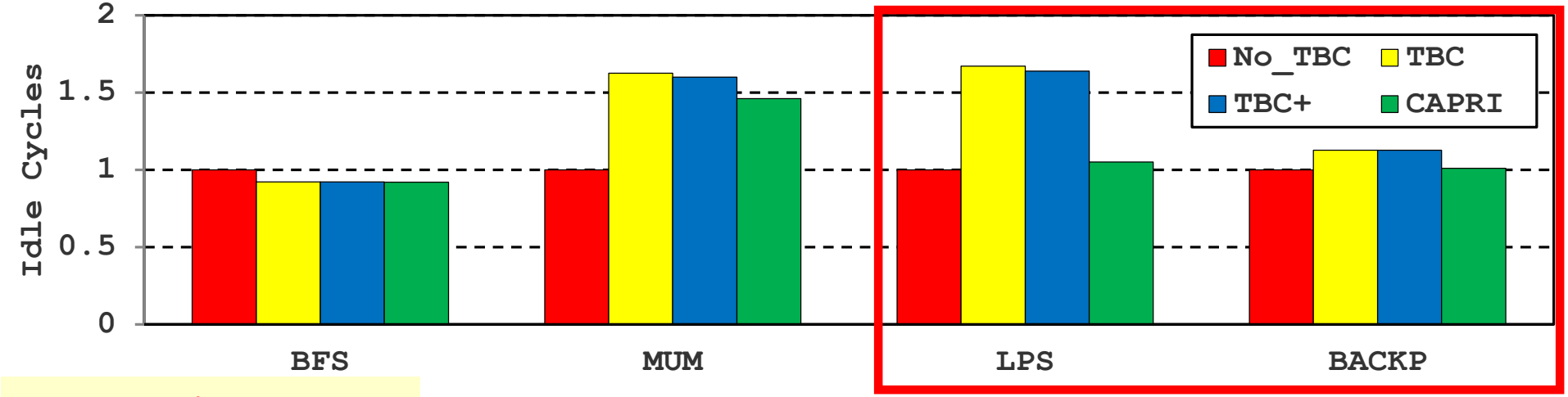
* Average number of SIMD lanes occupied when a warp is issued

Average SIMD lane utilization*



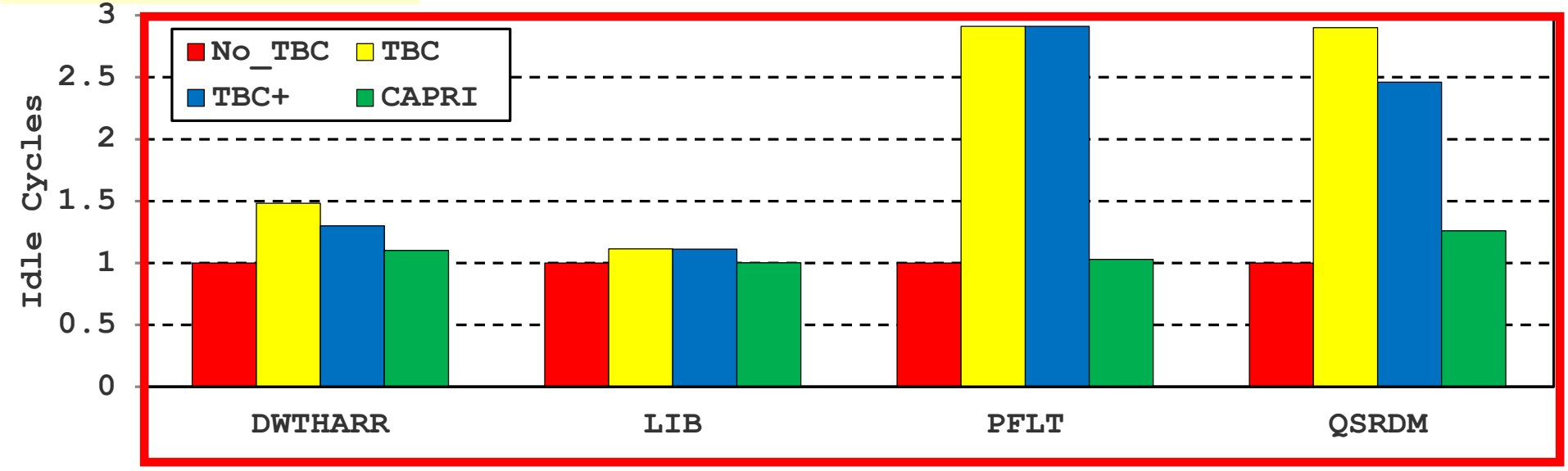
* Average number of SIMD lanes occupied when a warp is issued

Idle cycles (normalized)



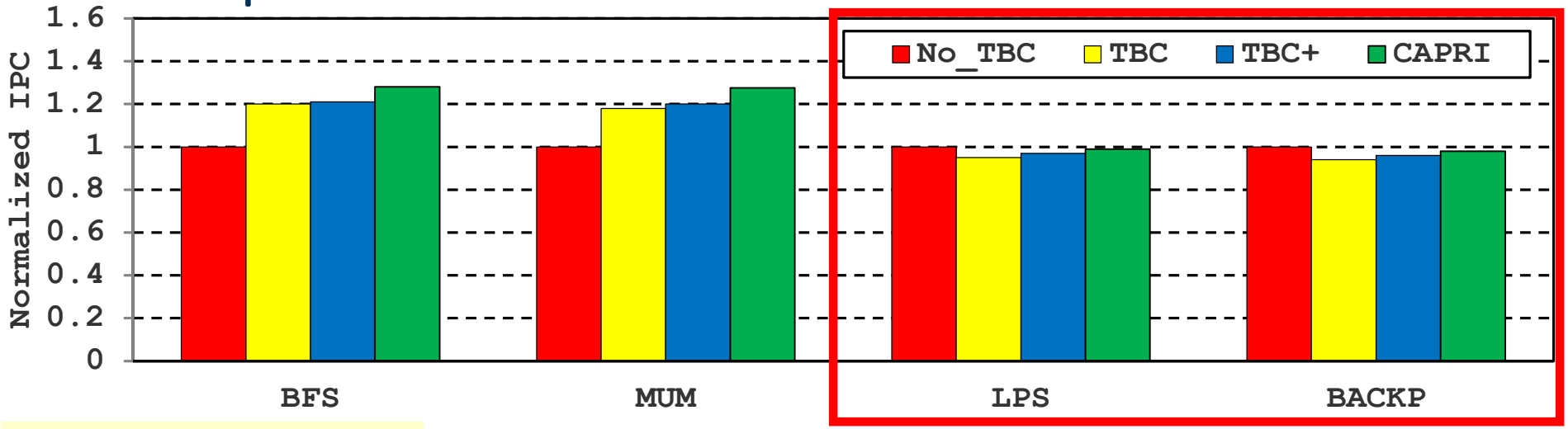
(a) Divergent Benchmarks

Lower is better



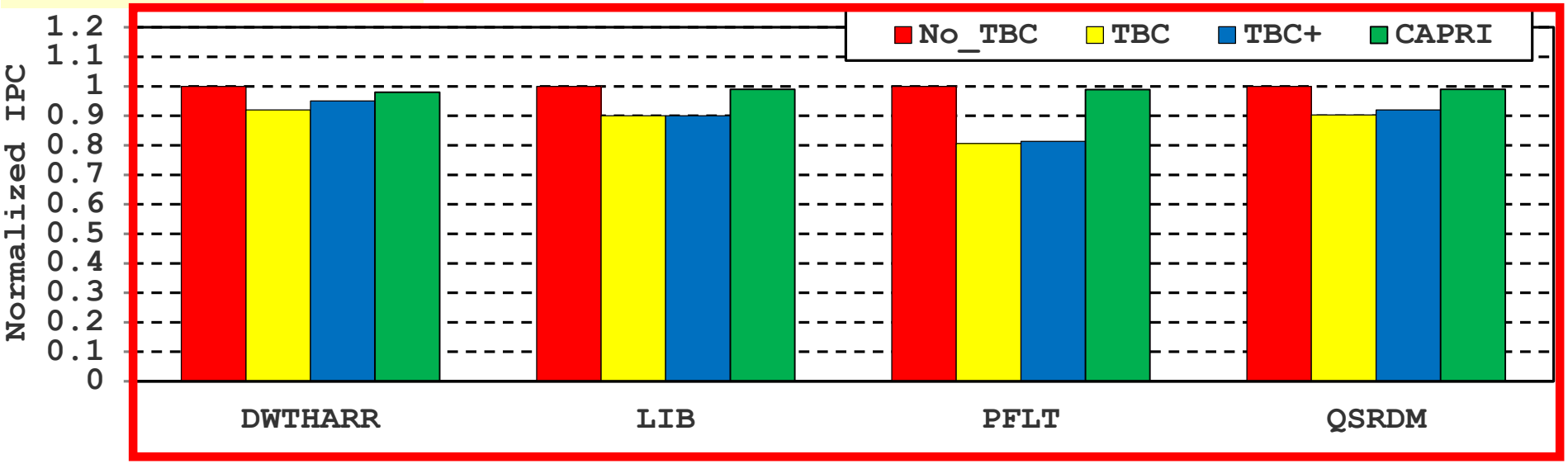
(b) Non-divergent Benchmarks

Overall performance



(a) Divergent Benchmarks

Higher is better

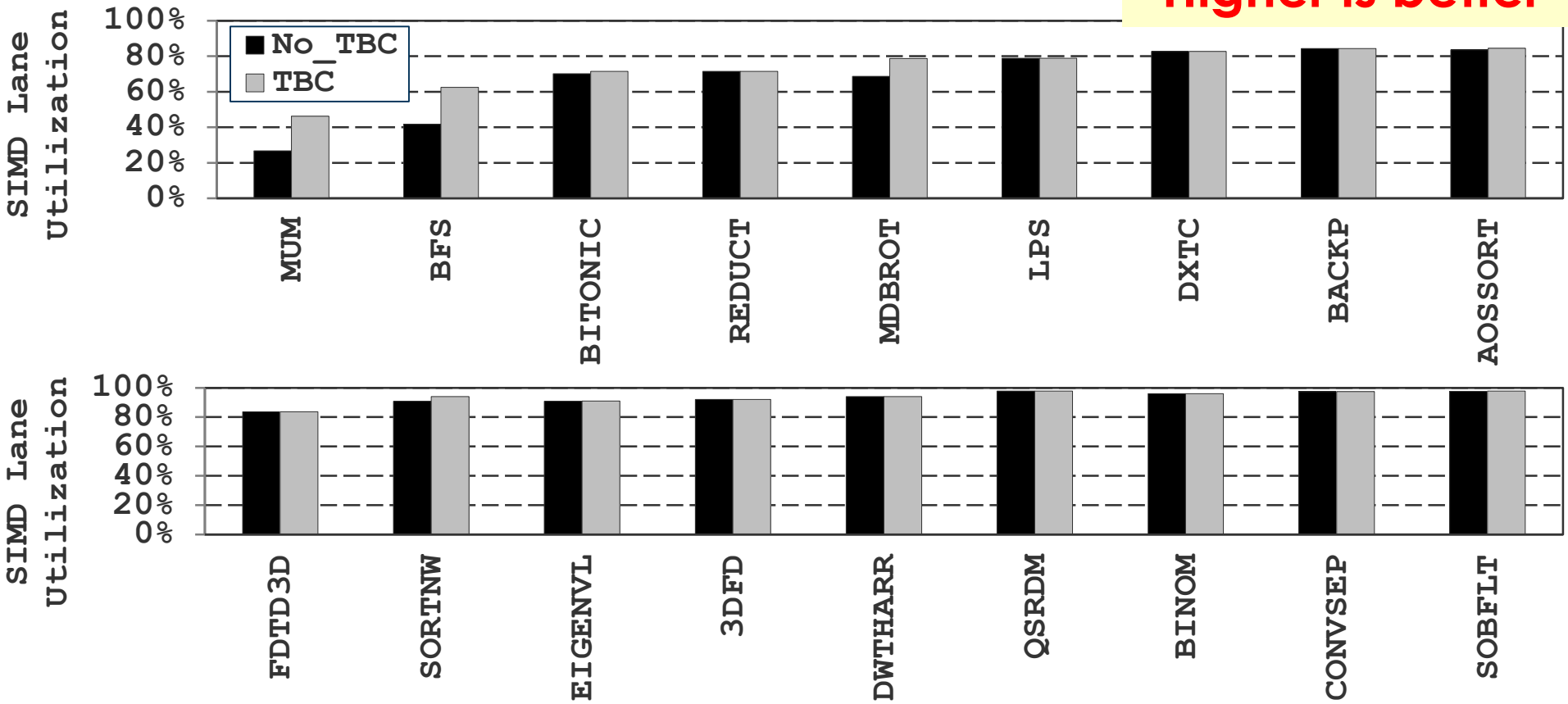


(b) Non-divergent Benchmarks

Limited applicability of compaction

- Average SIMD lanes occupied for execution
  - No_TBC : Baseline architecture without compaction
  - TBC : baseline compaction mechanism
    - Threads *maintain* execution in its home SIMD lan

Higher is better

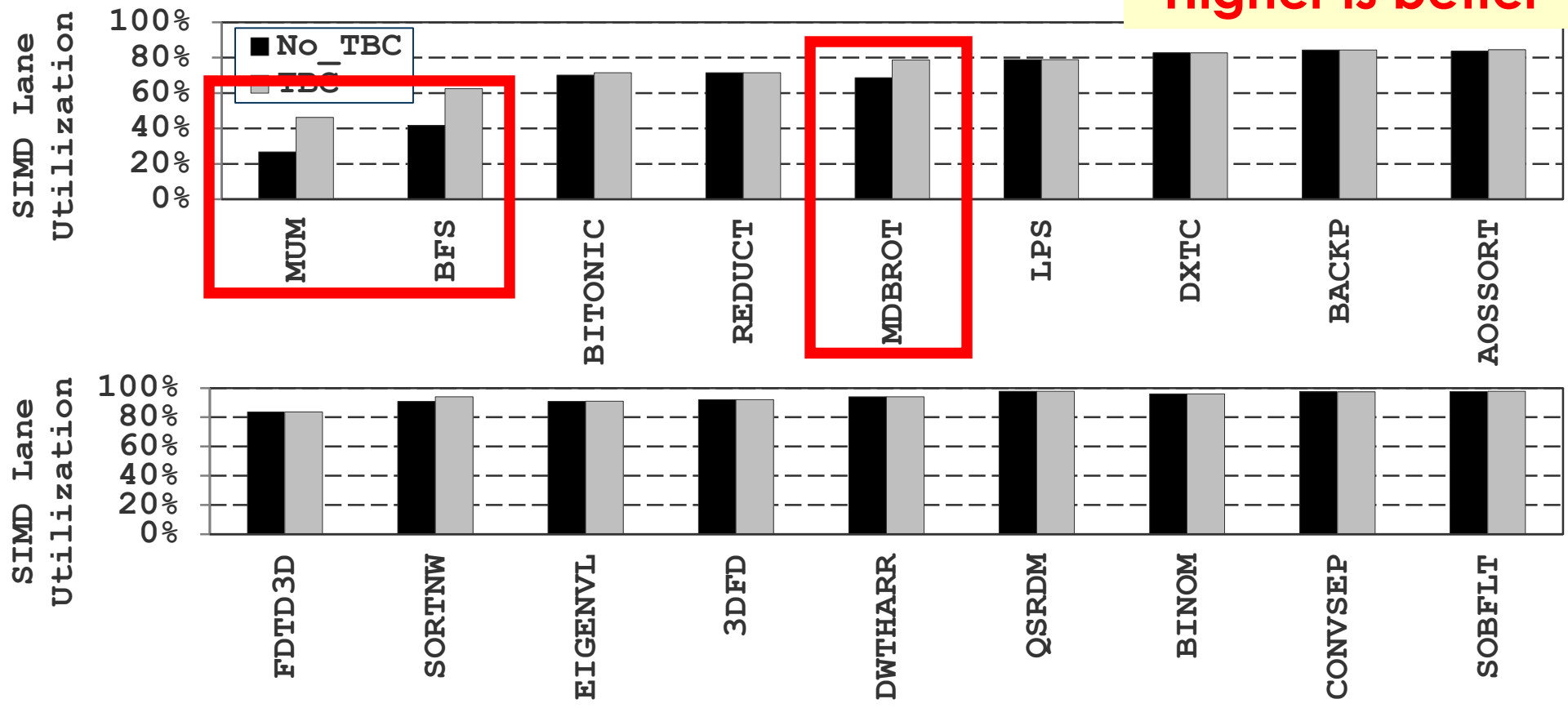


Limited applicability of compaction

- Average SIMD lanes occupied for execution

TBC, in general, only effective for *highly divergent* applications

Higher is better



Why does compaction fail?

- Most (or all) of SIMD lanes *already* occupied
  - Compaction inherently impossible

Active Threads	
W_0	0123
W_1	----
W_2	89AB
W_3	----

Non-divergent

Active Threads	
W_0	0123
W_1	456-
W_2	89AB
W_3	CDE-

Active Threads	
W_0	0123
W_1	4567
W_2	89AB
W_3	CD--

Divergent, but non-compactable

Why does compaction fail?

- Most (or all) of SIMD lanes *already* occupied
  - Compaction inherently impossible

Active Threads	
W_0	0123
W_1	----
W_2	89AB
W_3	----

Active Threads	
W_0	0123
W_1	456-
W_2	89AB
W_3	CDE-

Active Threads	
W_0	0123
W_1	4567
W_2	89AB
W_3	CD--

- Active threads ***aligned (clustered)*** on certain lanes
  - Compaction theoretically feasible, if crossbars can allow threads to execute in different SIMD lanes

Active Threads	
W_0	0---
W_1	4---
W_2	8---
W_3	C---

Active Threads	
W_0	01--
W_1	45--
W_2	89--
W_3	CD--

Active Threads	
W_0	0-2-
W_1	4-6-
W_2	8-A-
W_3	C-E-

Why does compaction fail?

- Most (or all) of SIMD lanes *already* occupied
  - Compaction inherently impossible

Active Threads	
W_0	0123
W_1	----
W_2	89AB
W_3	----

Active Threads	
W_0	0123
W_1	456-
W_2	89AB
W_3	CDE-

Active Threads	
W_0	0123
W_1	4567
W_2	89AB
W_3	CD--

- Active threads ***aligned (clustered)*** on certain lanes
  - Compaction theore ds to execute in different SIMD lanes

Aligned Divergence!

Active Threads	
W_0	0---
W_1	4---
W_2	8---
W_3	C---

Active Threads	
W_0	01--
W_1	45--
W_2	89--
W_3	CD--

Active Threads	
W_0	0-2-
W_1	4-6-
W_2	8-A-
W_3	C-E-

Aligned divergence – (1)

- Case 1: Branch condition depends on ***data*** array
  - Each thread references ***different*** element of the array
  - Unlikely to cause aligned divergence

Code #1) Branch depending on data arrays

```
0     // Code snippet from the kernel of BFS benchmark
1     // g_graph_visited and g_graph_edges are data array parameters.
2
3     int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
4
5     …
6     int id = g_graph_edges[…];
7     if( !g_graph_visited[id] )
8     {
9           …
10    }
```

Aligned divergence – (1)

- Case 1: Branch condition depends on ***data*** array
  - Each thread references ***different*** element of the array
  - Unlikely to cause aligned divergence

Code #1) Branch depending on data arrays

```
0     // Code snippet from the kernel of BFS benchmark
1     // g_graph_visited and g_graph_edges are data array parameters.
2
3     int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
4
5     …
6     int id = g_graph_edges[…];
7     if( !g_graph_visited[id] )
8     {
9           …
10    }
```

D-Branches

Aligned divergence – (2)

- Case 2: Branch cond. depends on ***programmatic*** value
  - *Indices* of thread-ID, warp-ID, CTA-ID, *width/height* of CTA
  - *Scalar* input parameters to the kernel, *constants*
  - Threads sharing home SIMD lane likely to reference ***same*** value

Code #2) Programmatic branch causing only the 1st half of the warp active

```
0     // Code snippet from the kernel of BACKP benchmark
1     // CTA is a (8 × 16) 2-D array of threads.
2
3     int tx = threadIdx.x;
4     int ty = threadIdx.y;
5     …
6     for (int i=1; i<=__log2f(HEIGHT); i++){
7        int power_two = __powf(2,i);
8
9        if( ty % power_two == 0 ) {…}
10       …
11    }
```

Aligned divergence – (2)

- Case 2: Branch cond. depends on ***programmatic*** value
  - *Indices* of thread-ID, warp-ID, CTA-ID, *width/height* of CTA
  - *Scalar* input parameters to the kernel, *constants*
  - Threads sharing home SIMD lane likely to reference ***same*** value

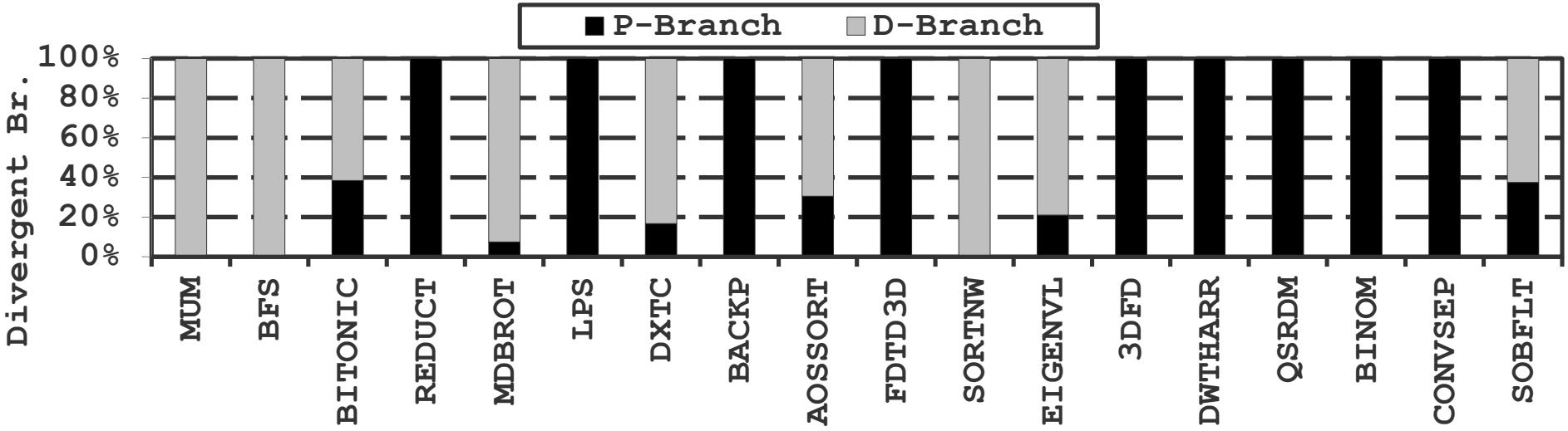
Code #2) Programmatic branch causing only the 1st half of the warp active

```
0    // Code snippet from the kernel of BACKP benchmark
1    // CTA is a (8 × 16) 2-D array of threads.
2
3    int tx = threadIdx.x;
4    int ty = threadIdx.y;
5    …
6    for (int i=1; i<=__log2f(HEIGHT); i++){
7       int power_two = __powf(2,i);
8
9       if( ty % power_two == 0 ) {…}
10      …
11   }
```

P-Branches

Compaction rate of p-/d-branches (with TBC)

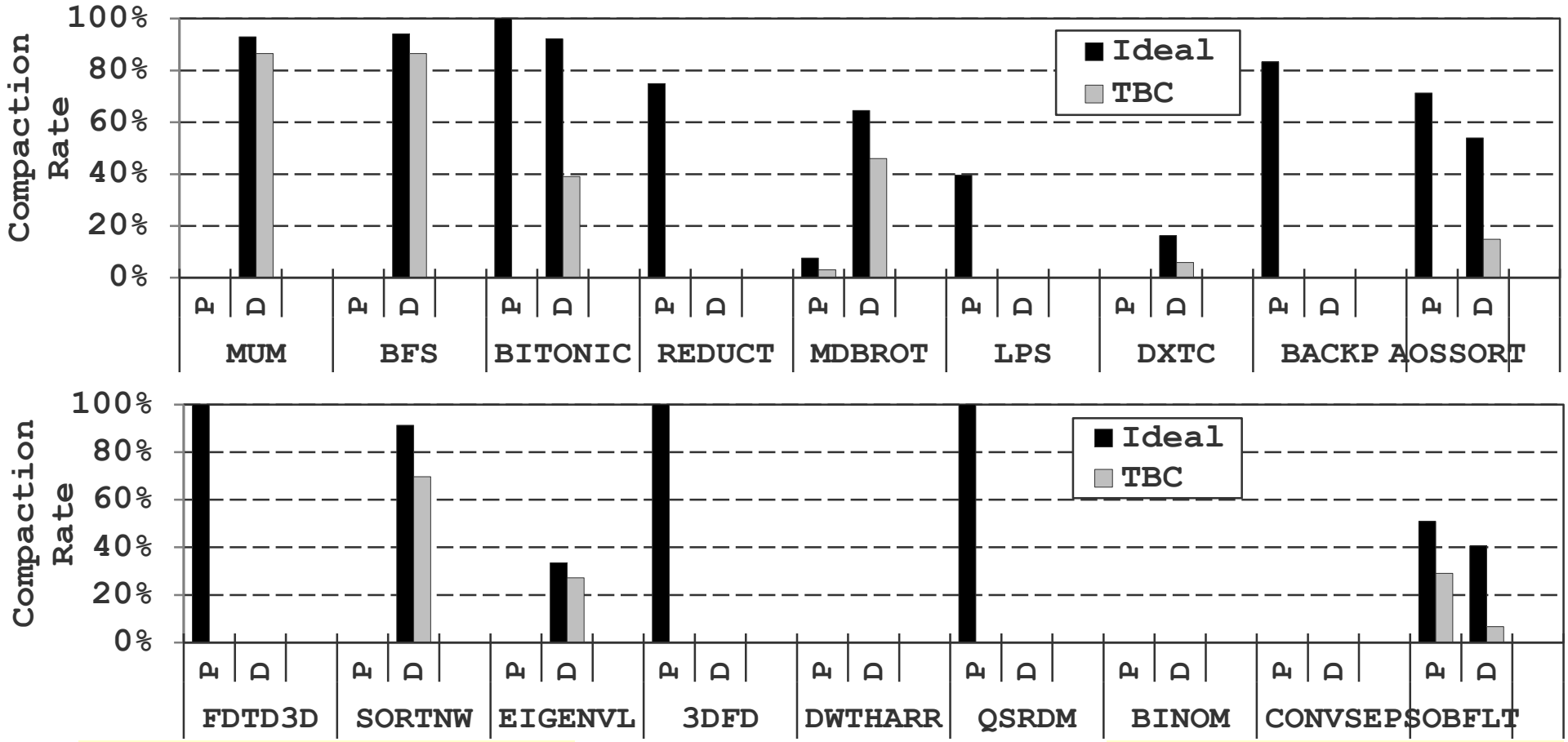
- Definition: Fraction of ***compactable*** paths among *all* paths generated by divergent branches



Branch categorization done with GPUOcelot using Taint-analysis (detailed in paper)

Compaction rate of p-/d-branches (with TBC)

- Definition: Fraction of ***compactable*** paths among *all* paths generated by divergent branches



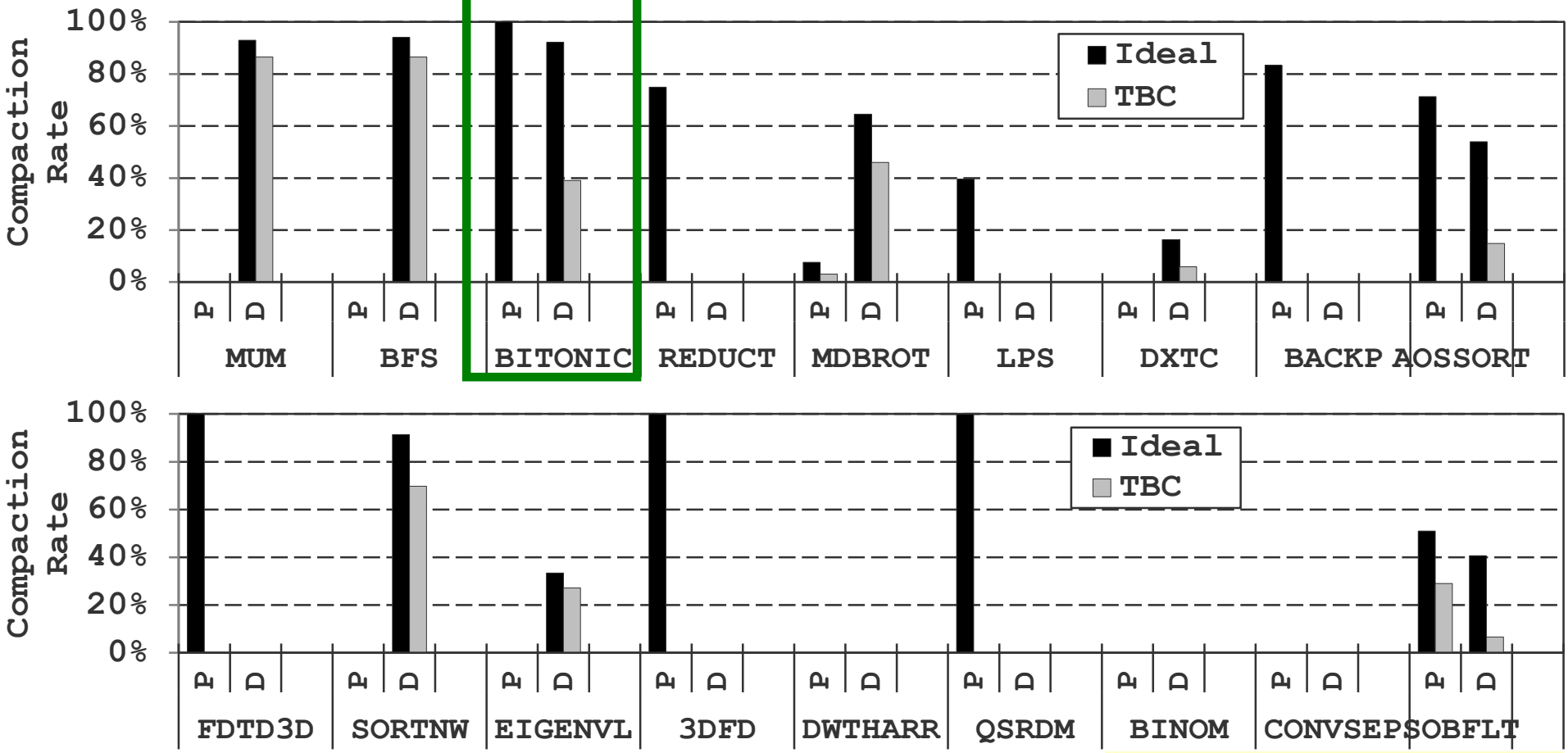
Ideal: TBC with crossbar

Higher is better

Compaction rate of p-/d-branches (with TBC)

D-branch compacts well with TBC
P-branch not so much compared to *Ideal*

all paths generated by divergent branches



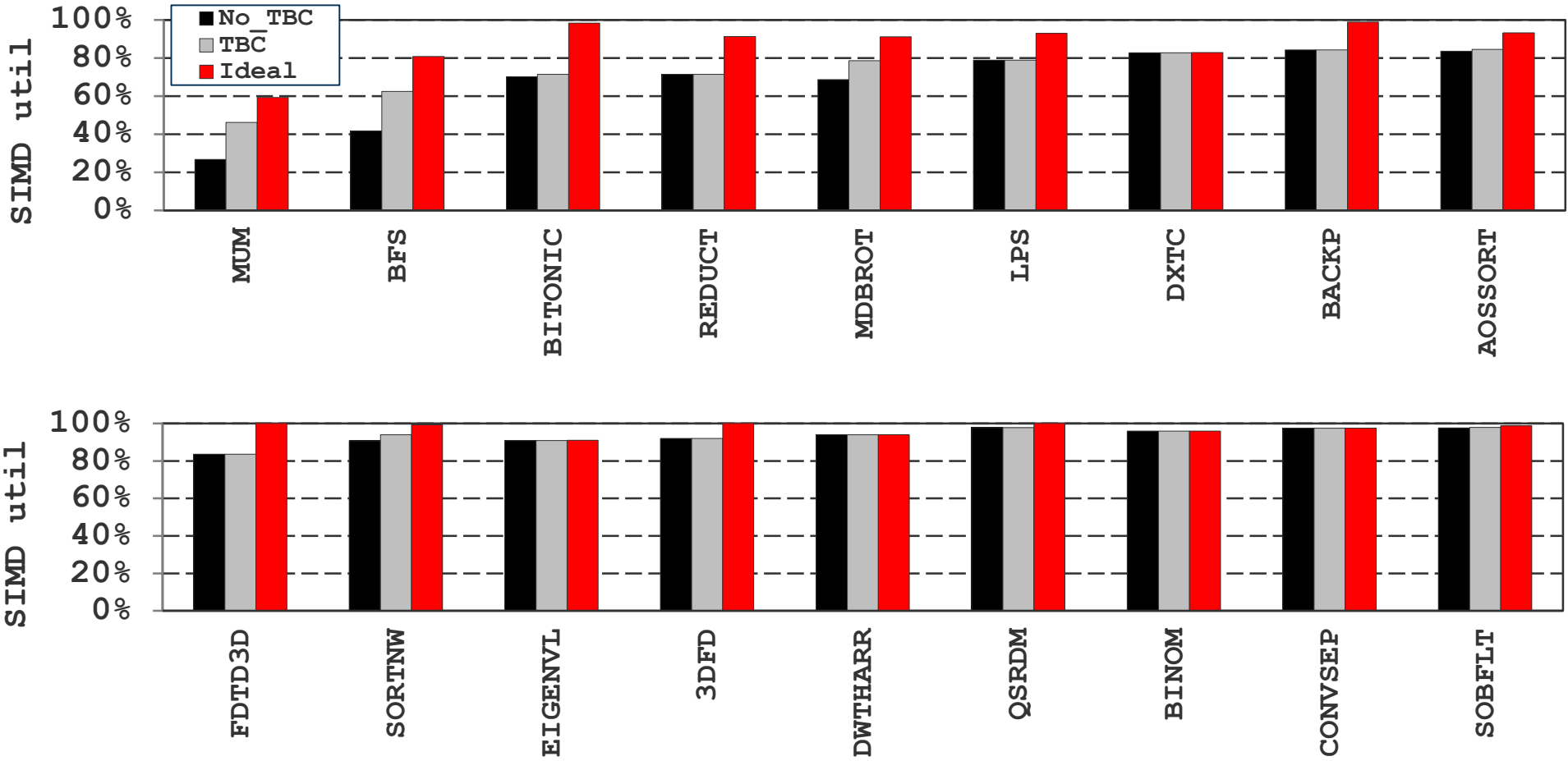
Ideal: TBC with crossbar

Higher is better

TBC *with* crossbar (*ideal* compaction)

- Average SIMD lanes occupied for execution

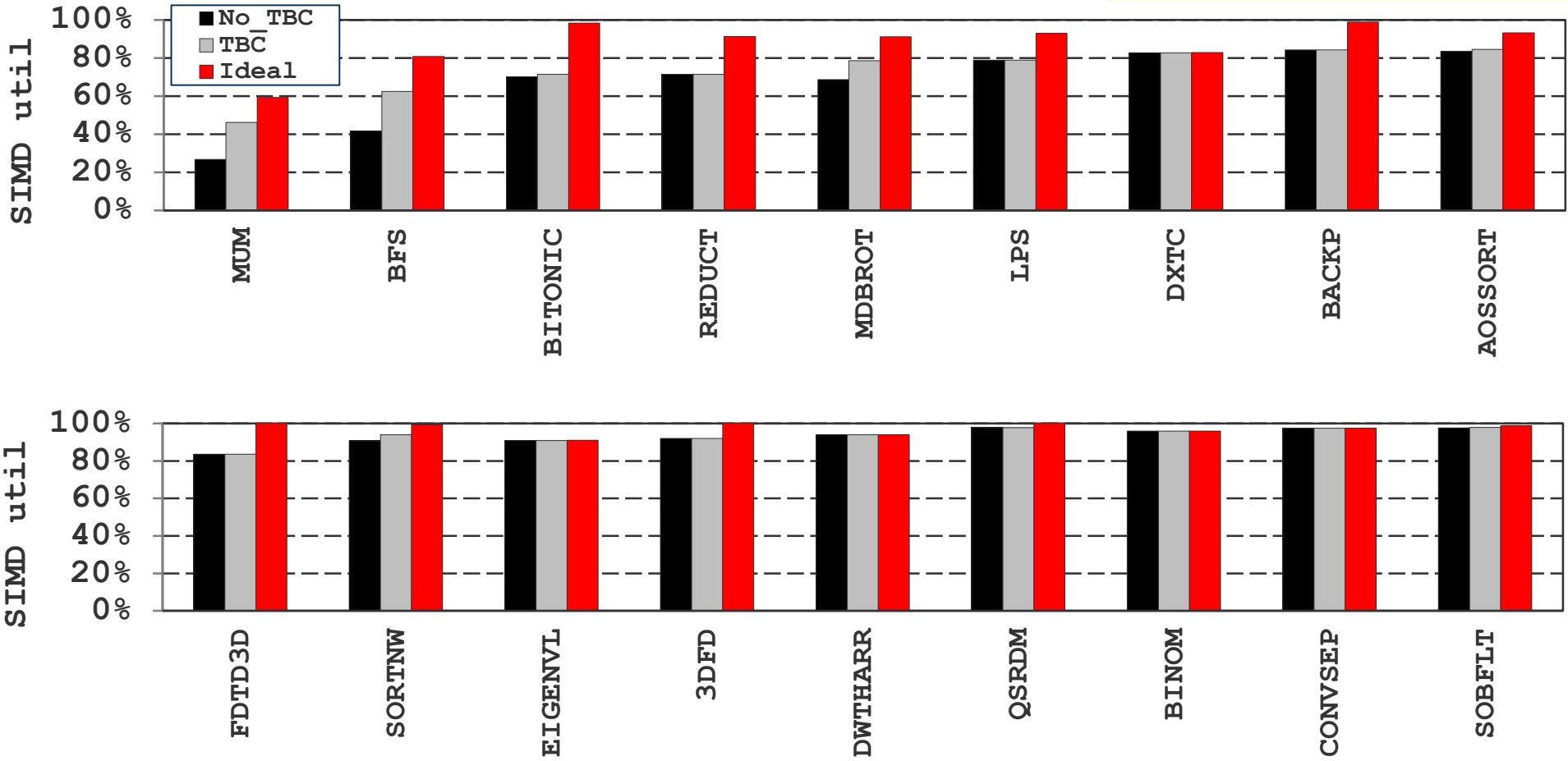
Higher is better



Significant *opportunities* to enhance SIMD efficiency by tackling aligned divergence.

But crossbars are *expensive!*

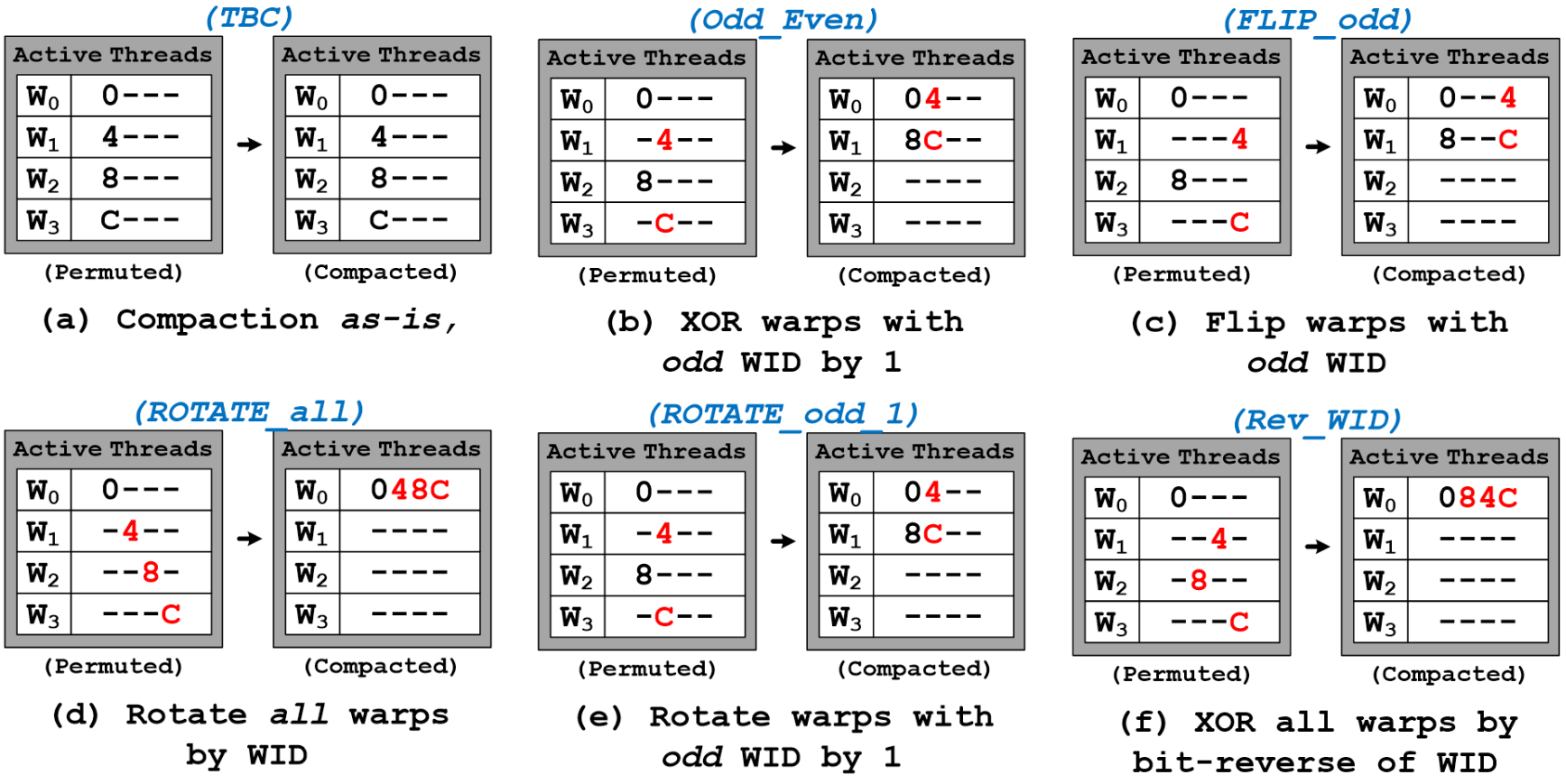
Higher is better



SIMD Lane Permutation (SLP)*

- **Motivation:** Permute home SIMD lanes to alleviate aligned divergence

* WID: Warp-ID



* *Rhu et al., "Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation", ISCA-2013*

Balanced permutation

- **Observation**: Divergence frequently exhibits a ***highly skewed*** concentration of active lanes
- **Intuition**: **Evenly balance** active threads across all lanes

Lane-ID	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
	000	001	010	011	100	101	110	111
For W0	0	1	2	3	4	5	6	7
For W1	0	1	2	3	4	5	6	7
For W2	0	1	2	3	4	5	6	7
For W3	0	1	2	3	4	5	6	7
For W4	0	1	2	3	4	5	6	7
For W5	0	1	2	3	4	5	6	7
For W6	0	1	2	3	4	5	6	7
For W7	0	1	2	3	4	5	6	7

Balanced permutation

- **Observation**: Divergence frequently exhibits a ***highly skewed*** concentration of active lanes
- **Intuition**: **Evenly balance** active threads across all lanes

Lane-ID	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
	000	001	010	011	100	101	110	111
For W0	0	1	2	3	4	5	6	7
For W1	0	1	2	3	4	5	6	7
For W2	0	1	2	3	4	5	6	7
For W3	0	1	2	3	4	5	6	7
For W4	0							
For W5	0							
For W6	0	1	2	3	4	5	6	7
For W7	0	1	2	3	4	5	6	7

Baseline: assigns lane-IDs based on *round-robin* manner

Balanced permutation

- **Observation**: Divergence frequently exhibits a ***highly skewed*** concentration of active lanes
- **Intuition**: **Evenly balance** active threads across all lanes

Lane-ID	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
	000	001	010	011	100	101	110	111
For W0	0	1	2	3	4	5	6	7
For W1	0	1	2	3	4	5	6	7
For W2	0	1	2	3	4	5	6	7
For W3	0	1	2	3	4	5	6	7
For W4	0	1	2	3	4	5	6	7
For W5	0	1	2	3	4	5	6	7
For W6	0	1	2	3	4	5	6	7
For W7	0	1	2	3	4	5	6	7

Balanced permutation

- Obse
- Intuit

< *Balanced* permutation algorithm >
- Even-ID warps: *XOR* lane-ID by (Warp-ID >> 1)
- Odd-ID warps: *XOR* lane-ID by ~(Warp-ID >> 1)

Lane-ID	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
	000	001	010	011	100	101	110	111
For W0	0	1	2	3	4	5	6	7
For W1	0	1	2	3	4	5	6	7
For W2	0	1	2	3	4	5	6	7
For W3	0	1	2	3	4	5	6	7
For W4	0	1	2	3	4	5	6	7
For W5	0	1	2	3	4	5	6	7
For W6	0	1	2	3	4	5	6	7
For W7	0	1	2	3	4	5	6	7

Balanced permutation

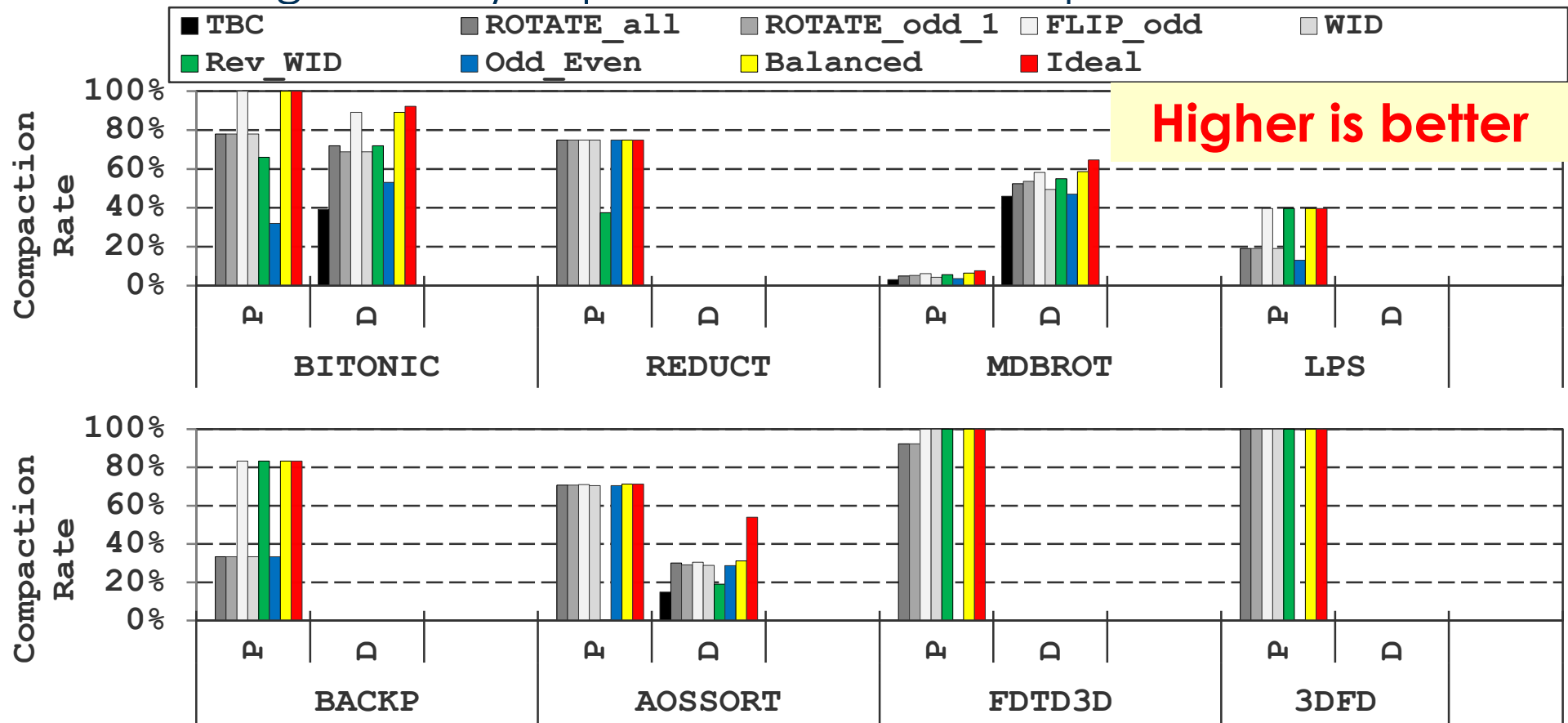
- Obse
- Intuit

< *Balanced* permutation algorithm >
- Even-ID warps: *XOR* lane-ID by (Warp-ID >> 1)
- Odd-ID warps: *XOR* lane-ID by ~(Warp-ID >> 1)

	Lane-ID	0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
For W0	XOR-000	0	1	2	3	4	5	6	7
For W1	XOR-111	7	6	5	4	3	2	1	0
For W2	XOR-001	1	0	3	2	5	4	7	6
For W3	XOR-110	6	7	4	5	2	3	0	1
For W4	XOR-010	2	3	0	1	6	7	4	5
For W5	XOR-101	5	4	7	6	1	0	3	2
For W6	XOR-011	3	2	1	0	7	6	5	4
For W7	XOR-100	4	5	6	7	0	1	2	3

Compaction rate

- **Definition**: Fraction of ***compactable*** paths among *all* paths generated by divergent branches
- **SLP**: significantly improves *P*-branch compaction rate

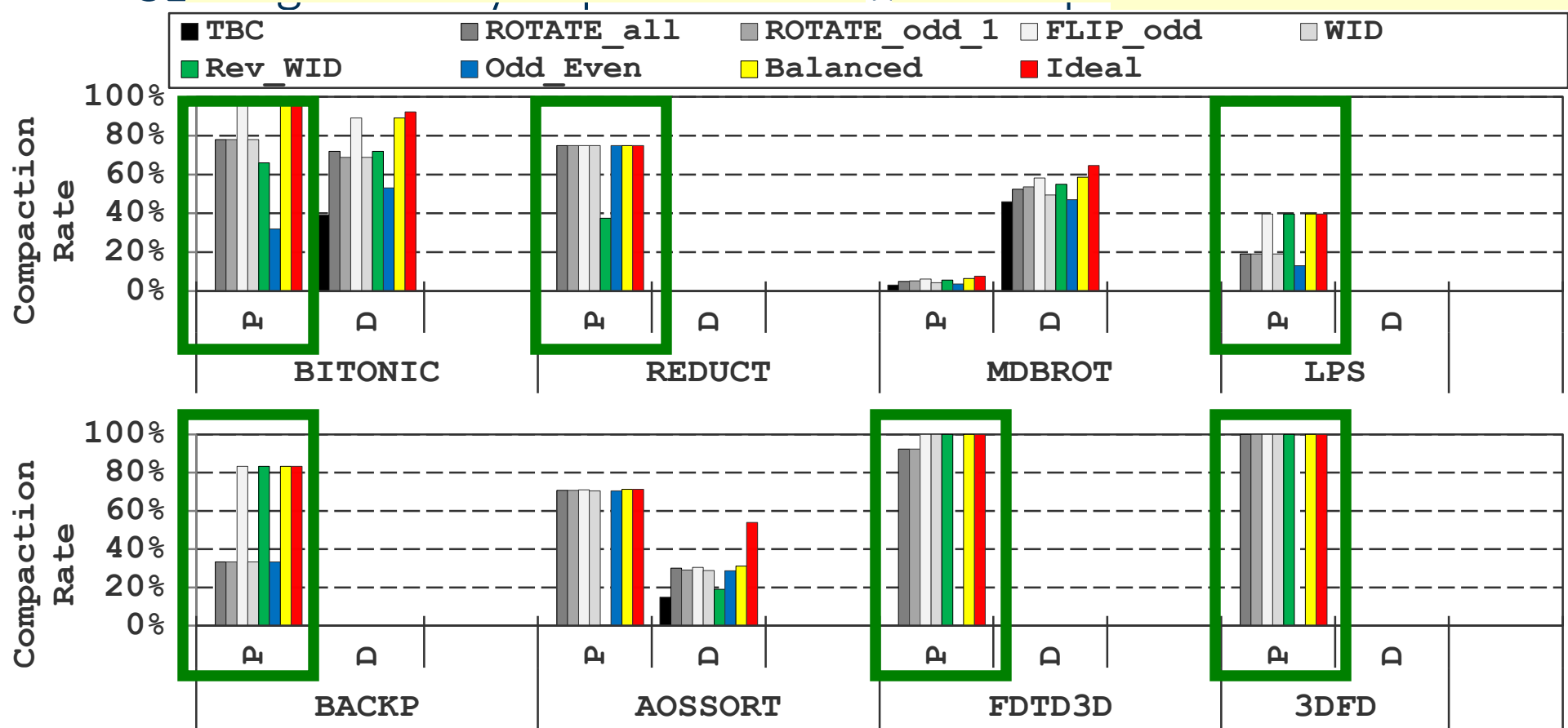


Compaction rate

- De
- SL

- **P-branches**
  - **TBC**: average 3.2%
  - **Odd_Even**: average 28.9%
  - **Balanced**: average 71.5%
  - **Ideal**: average 72.7%

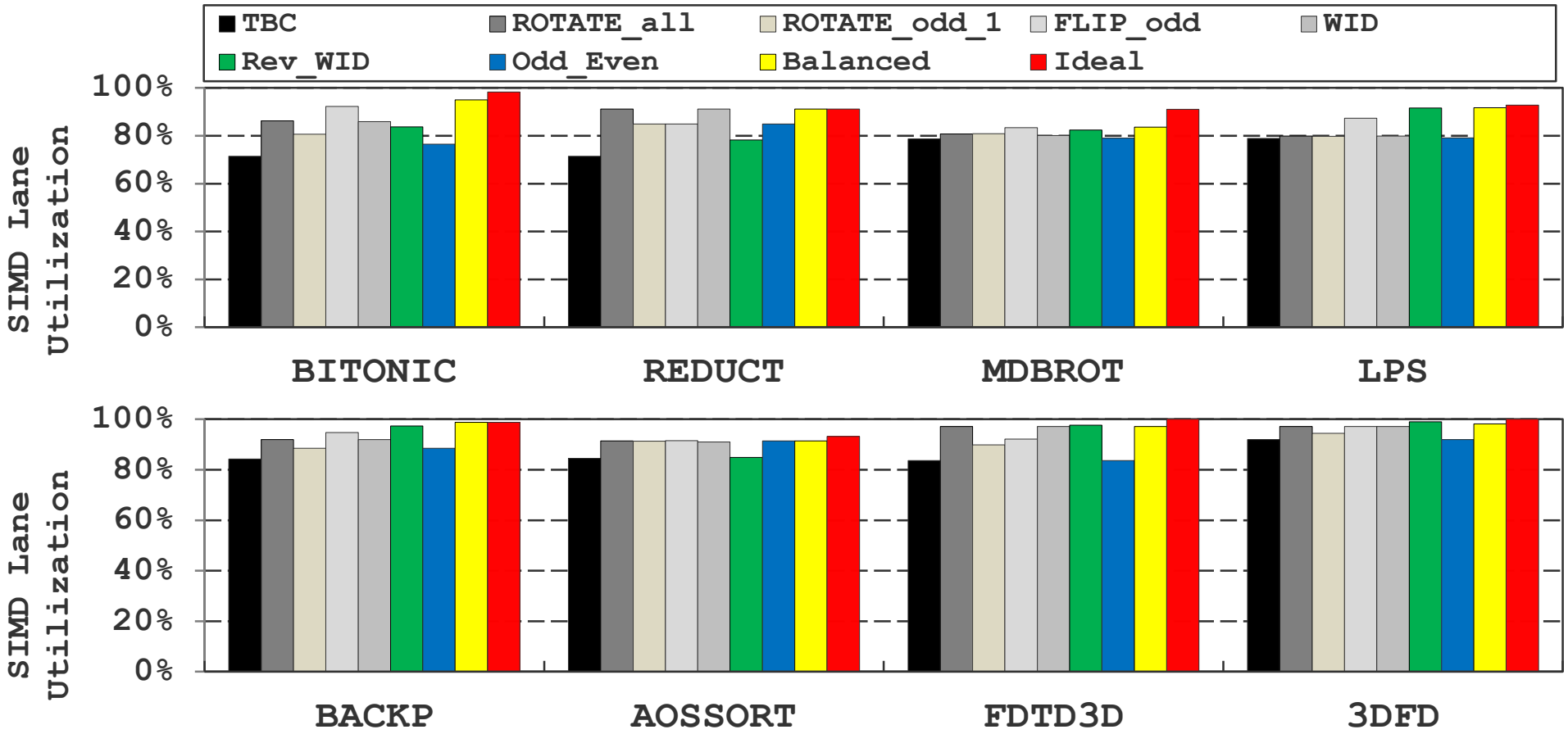
- **D-branches**
  - **TBC**: average 42.5%
  - **Odd_Even**: average 45.2%
  - **Balanced**: average 59.3%
  - **Ideal**: average 68.5%



Average SIMD lane utilization

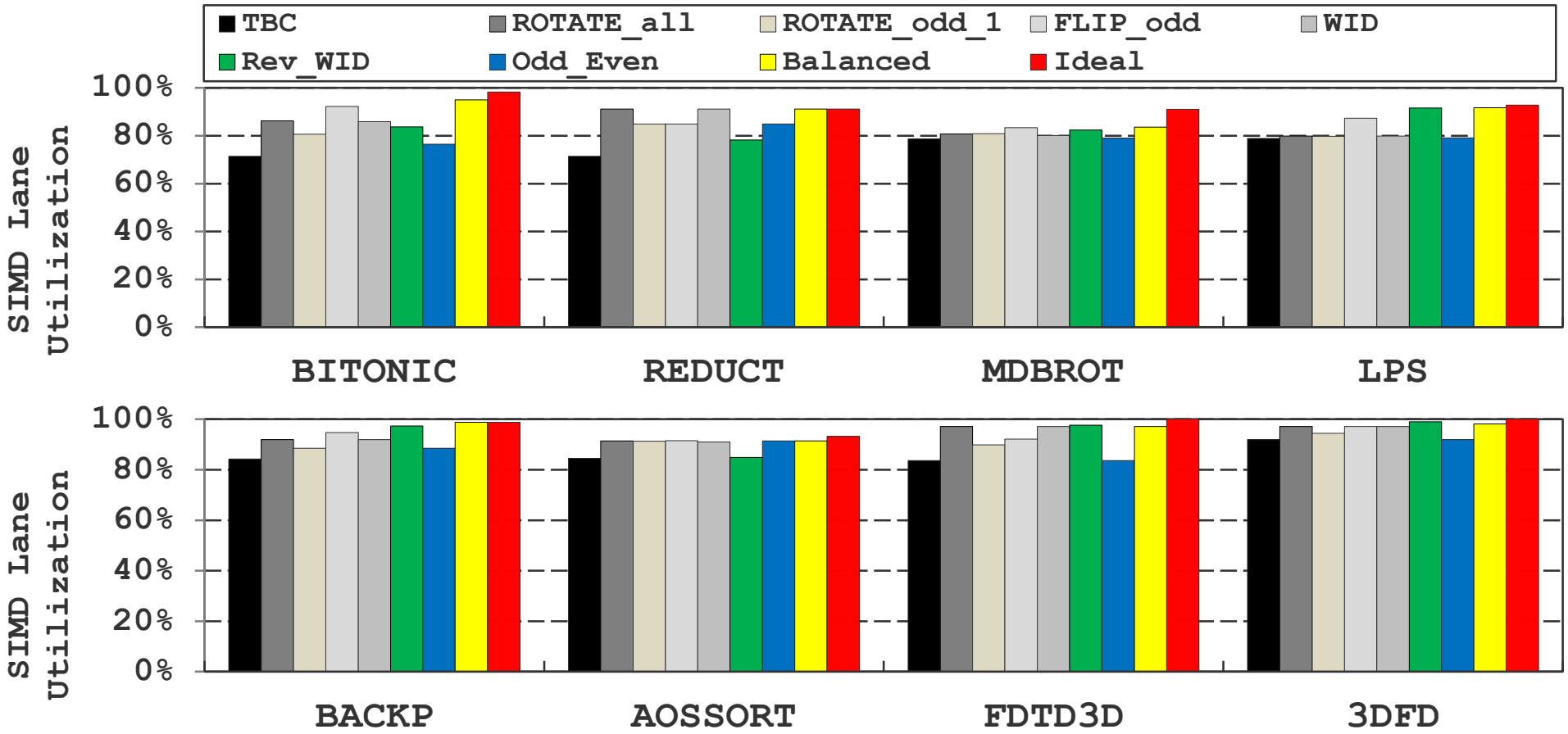
- **Definition**: average number of SIMD lanes *actually* executing and committing results

Higher is better



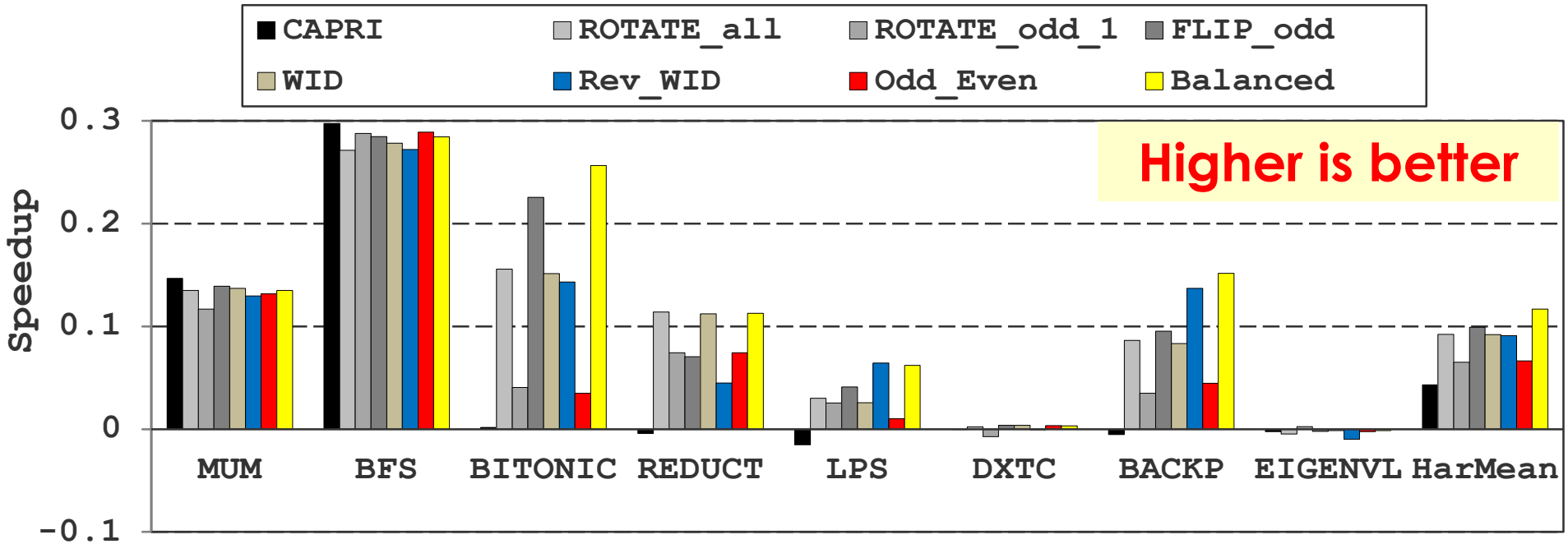
Average SIMD lane utilization

- **- TBC**: lowest utilization due to un-compacted aligned divergence
- **- SLP**:
  - ***Odd_Even***: average 2.1% (max 18%) increase over TBC
  - ***Balanced***: average 7.1% (max 34%) increase over TBC



Speedup

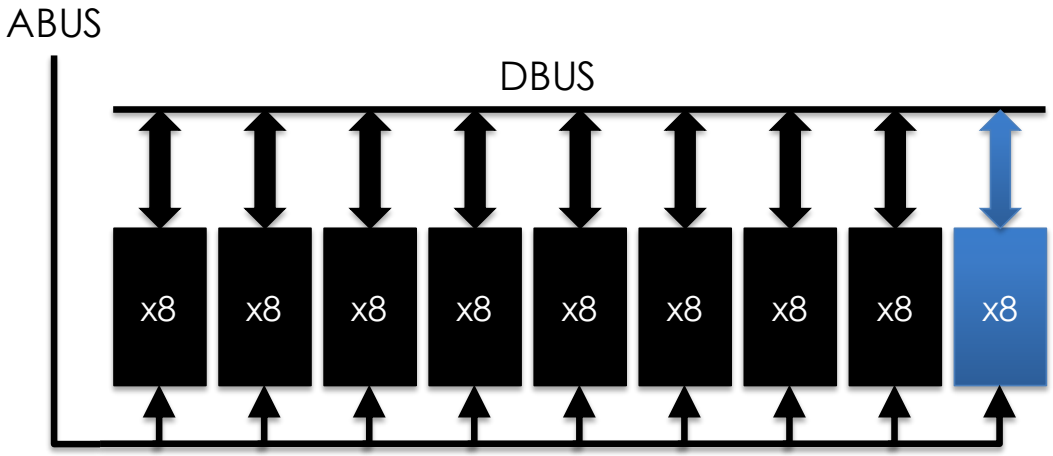
- **Baseline**: no compaction
- TBC+CAPRI only effective for 'irregular' applications
- SLP widens the range of applications that benefit from compaction techniques
  - 7.1% (max 34%) improvements on top of TBC+CAPRI



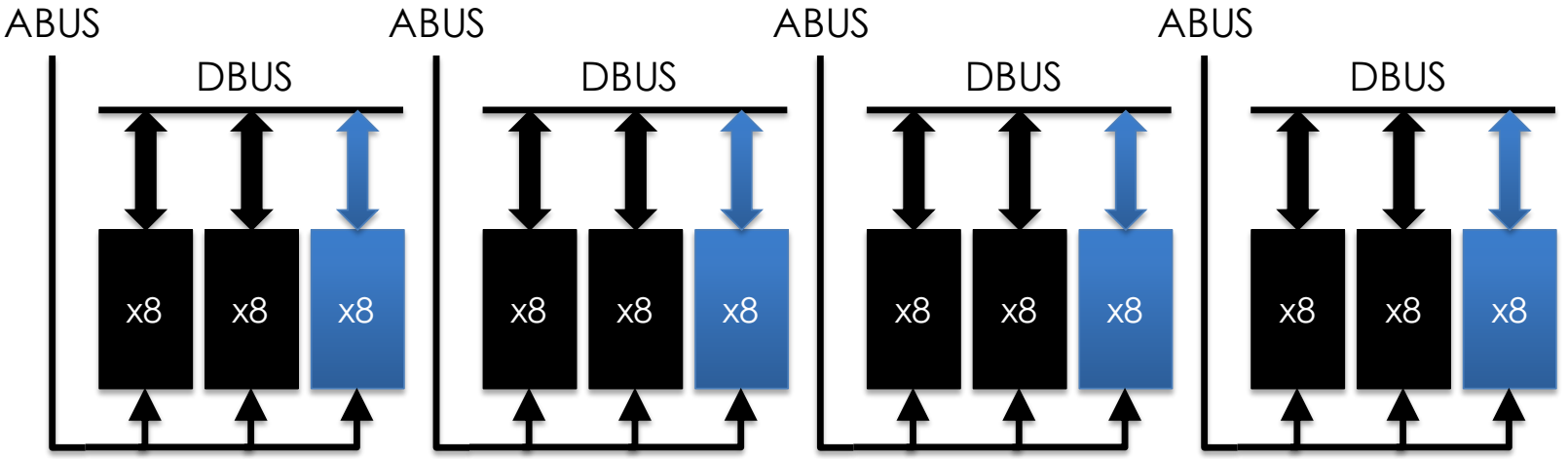
IRREGULAR AND FINE-GRAIN MEMORY ACCESS

CG-only and FG-only systems

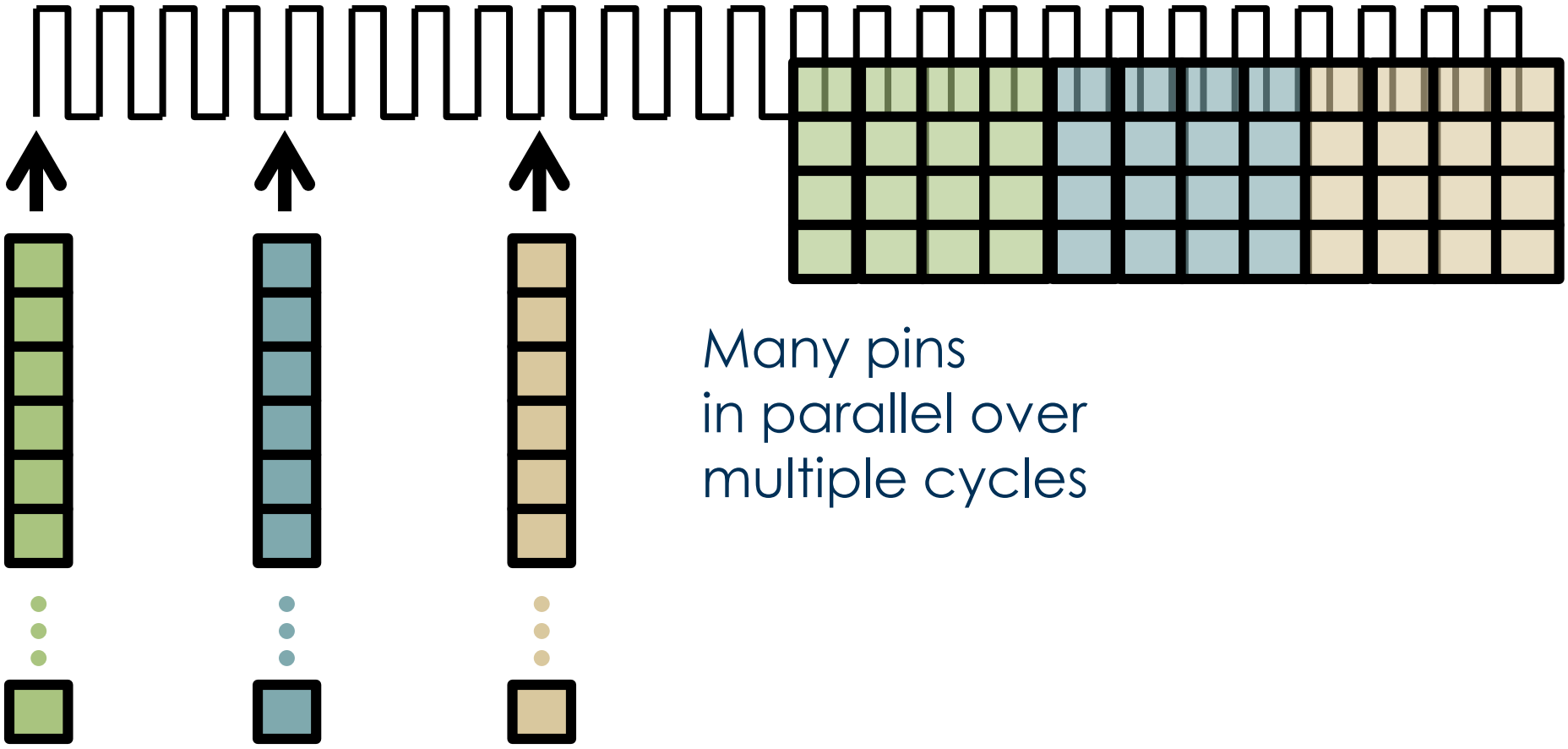
CG-Only: Wide DRAM channel



FG-Only: Many narrow channels

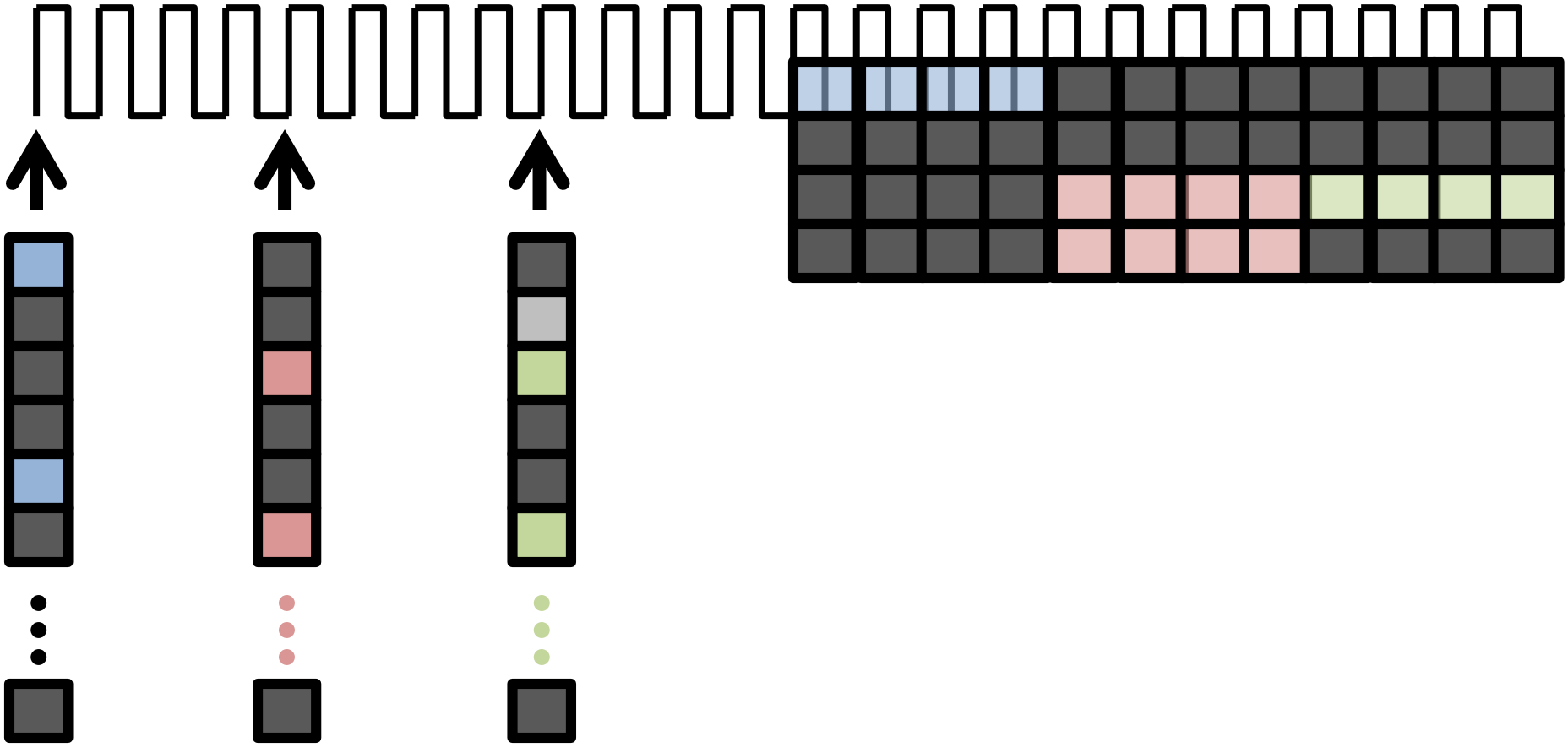


Latency + throughput + efficiency → **parallelism**



Hierarchy + parallelism

→ potential waste



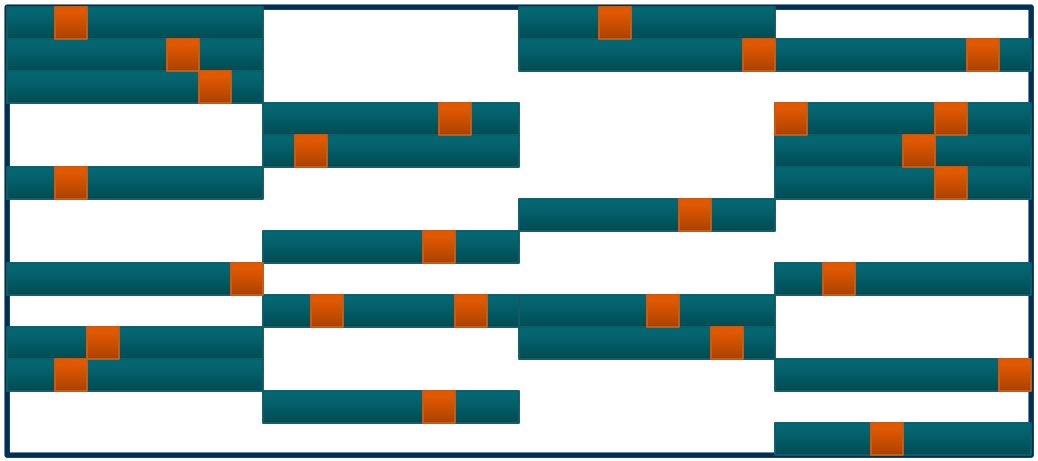
CG access MAY Waste BW

GUPS microbenchmark

```
for( i=0; i<N; i++ ) {
  a[ b[i] ] += x;
}
```

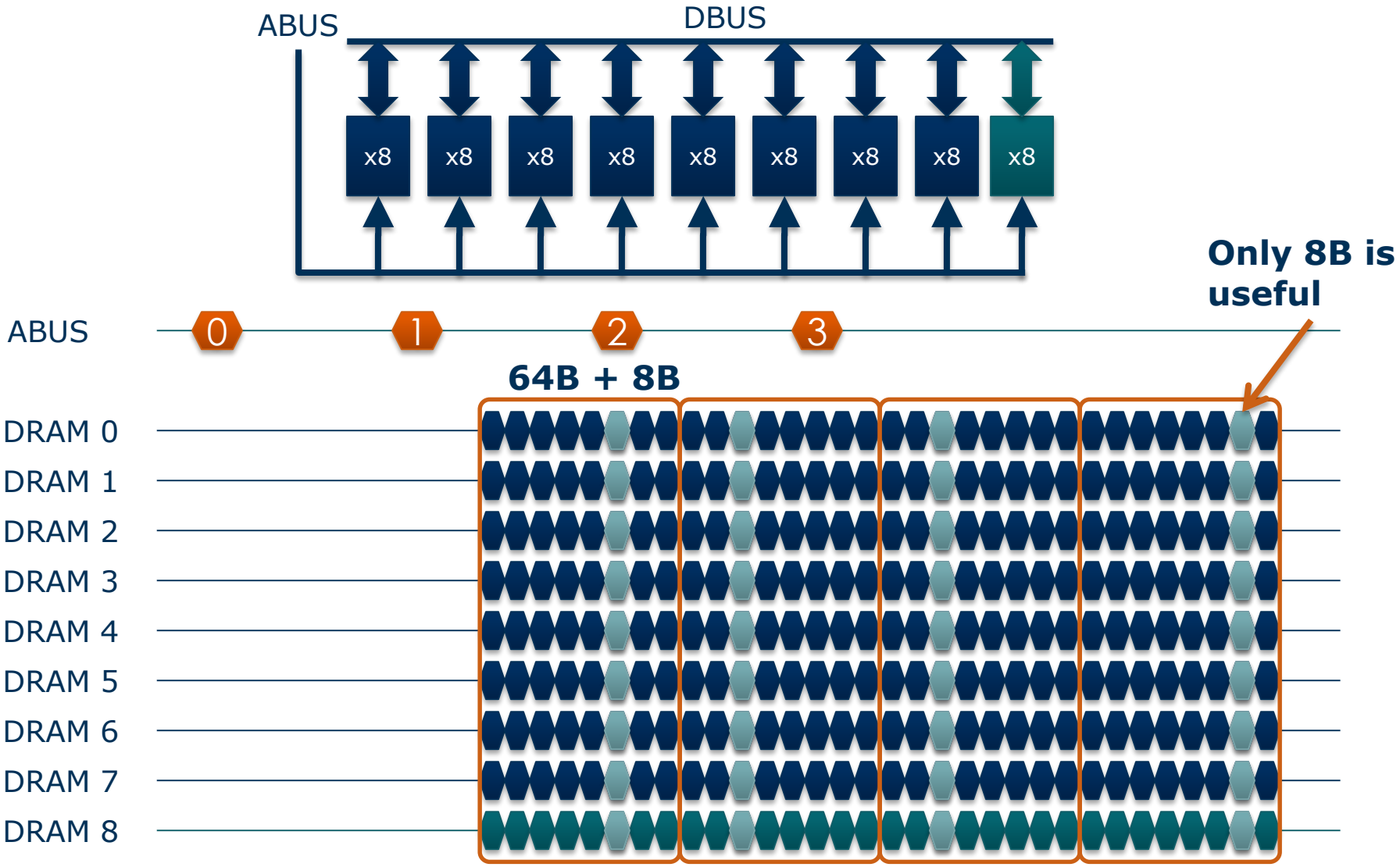
Initialized with random numbers

Buffer a

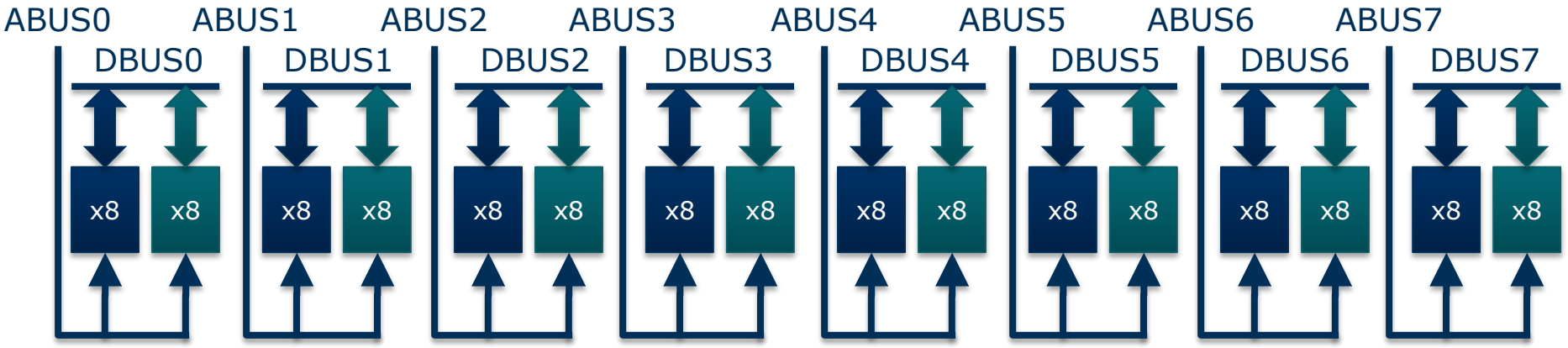


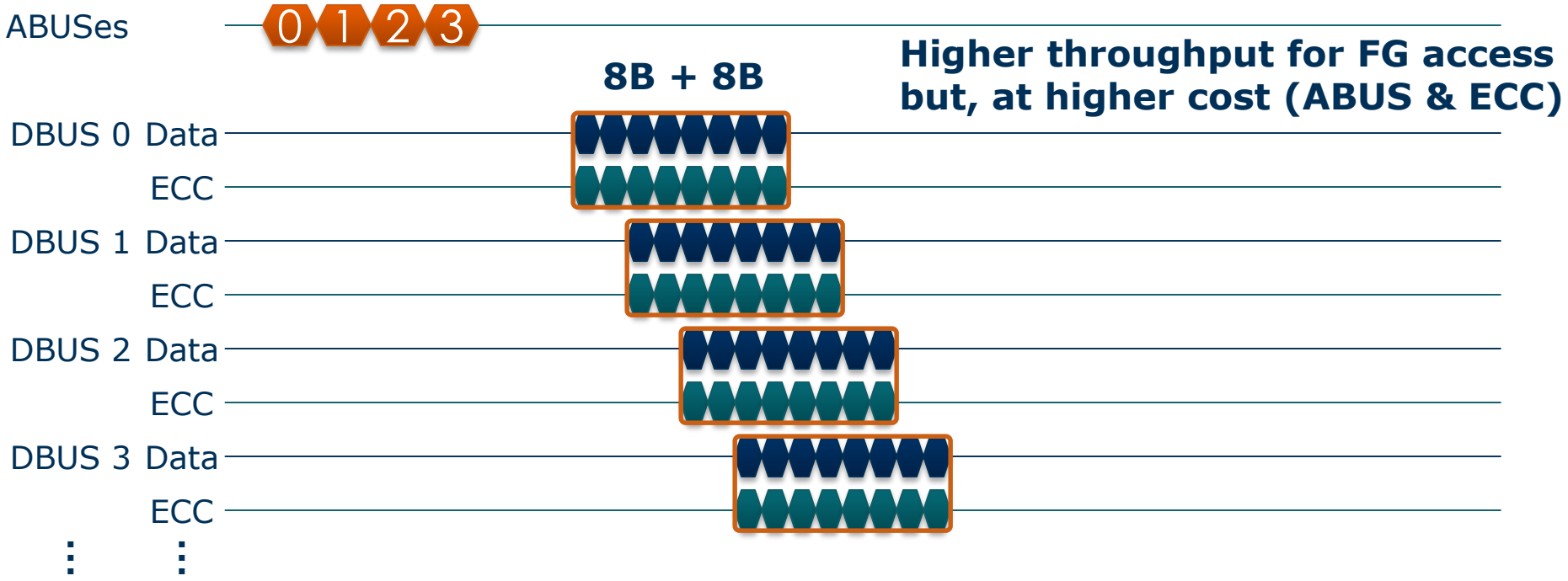
Waste BW on unused data

Conventional CG-only always retrieves 64B



FG-only is a very expensive solution

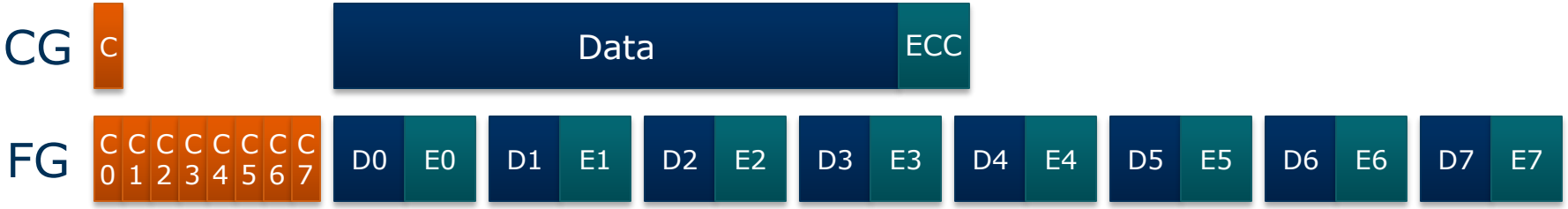




Higher throughput for FG access but, at higher cost (ABUS & ECC)

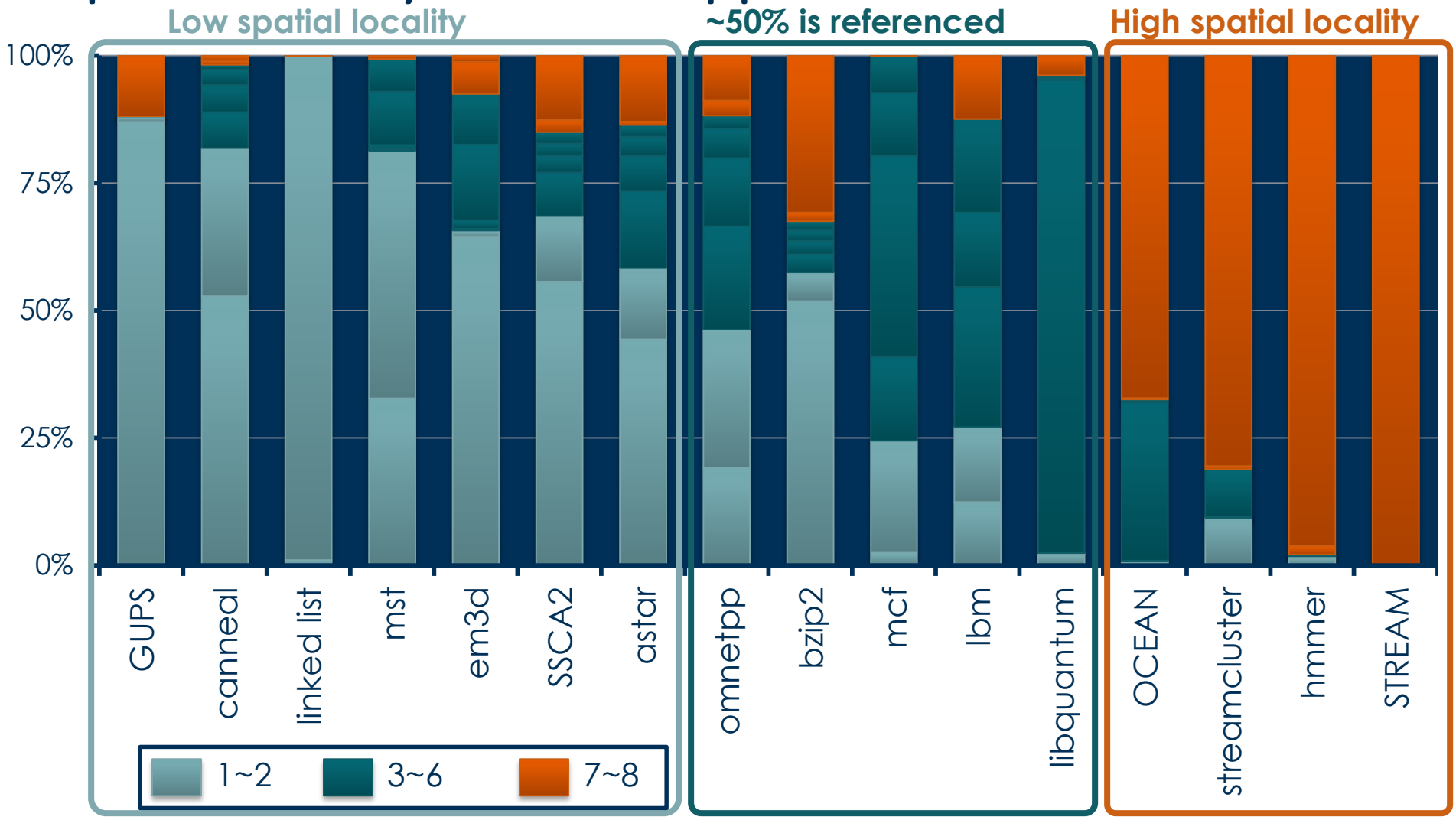
CG and FG Access

- Coarse-grained access
  - Control and ECC overheads are amortized over a large block
  - Waste BW when spatial locality is low



- Fine-grained access
  - Higher control and ECC overheads
  - Potentially higher throughput when spatial locality is low

Spatial locality in actual applications



PIN-based profiling with a 1MB cache with a 64B cache line

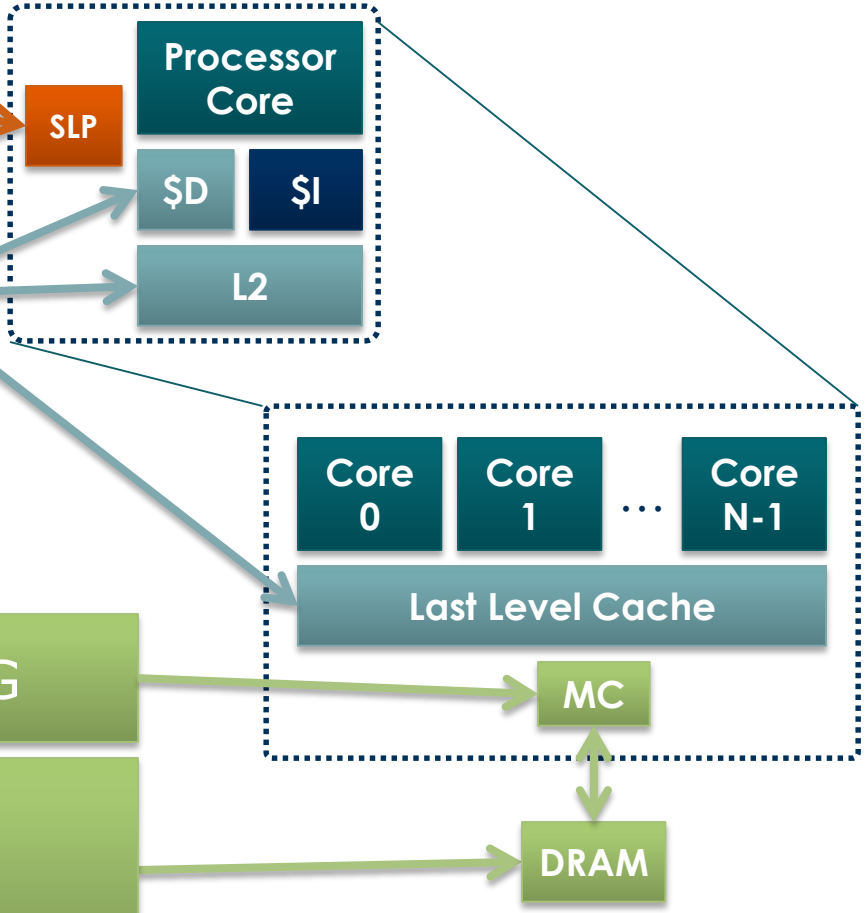
Dynamic granularity is best of both worlds

Locality Predictor

Sector cache
(8 8B sectors per line)

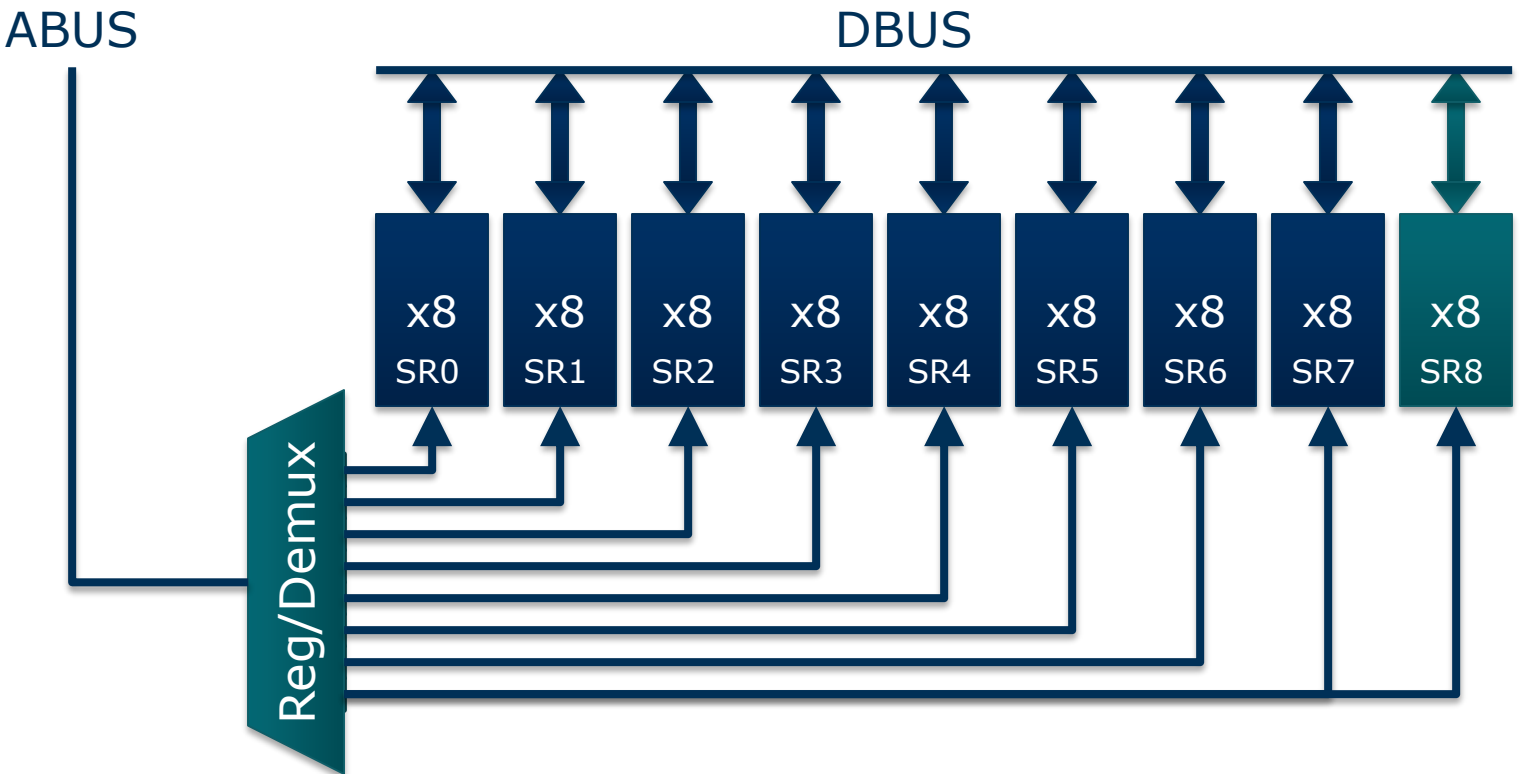
Co-scheduling CG/FG w/ split CG

Sub-Ranked DRAM w/ 2X ABUS
Support both CG and FG



Sub-ranked memory enables DGMS

- Control individual DRAM chips independently
  - Originally proposed (mostly) for energy efficiency w/ CG
    - Threaded module, MC-DIMM, Mini-rank, S/G DIMM
- Access granularity = 8B (8 bit x 8 burst)



Sector Cache

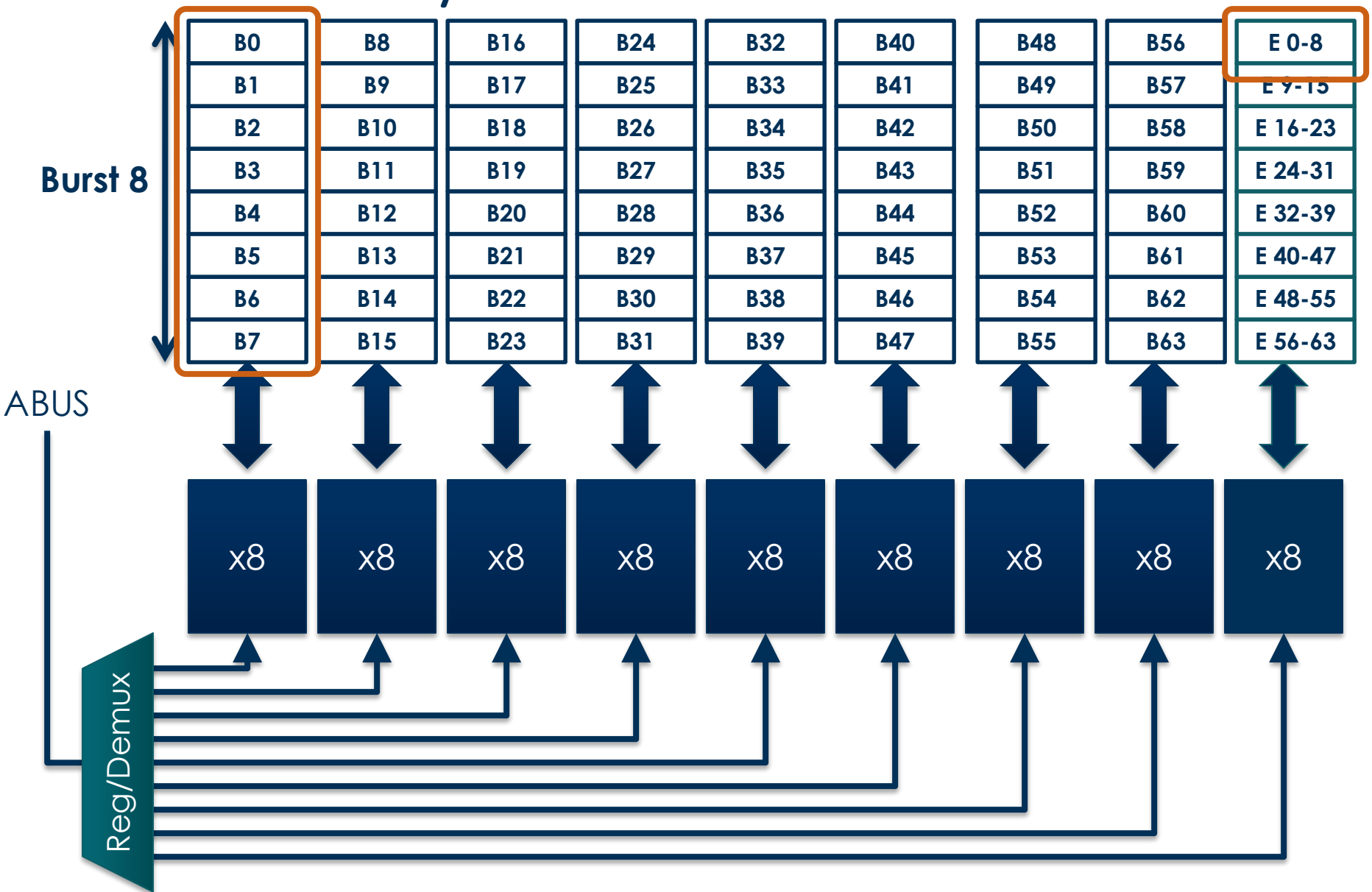
- Caches also need to manage fine-grained data
- We use a simple *sector* cache
  - Allow partially valid cache lines
  - Low tag overhead

Long cache line

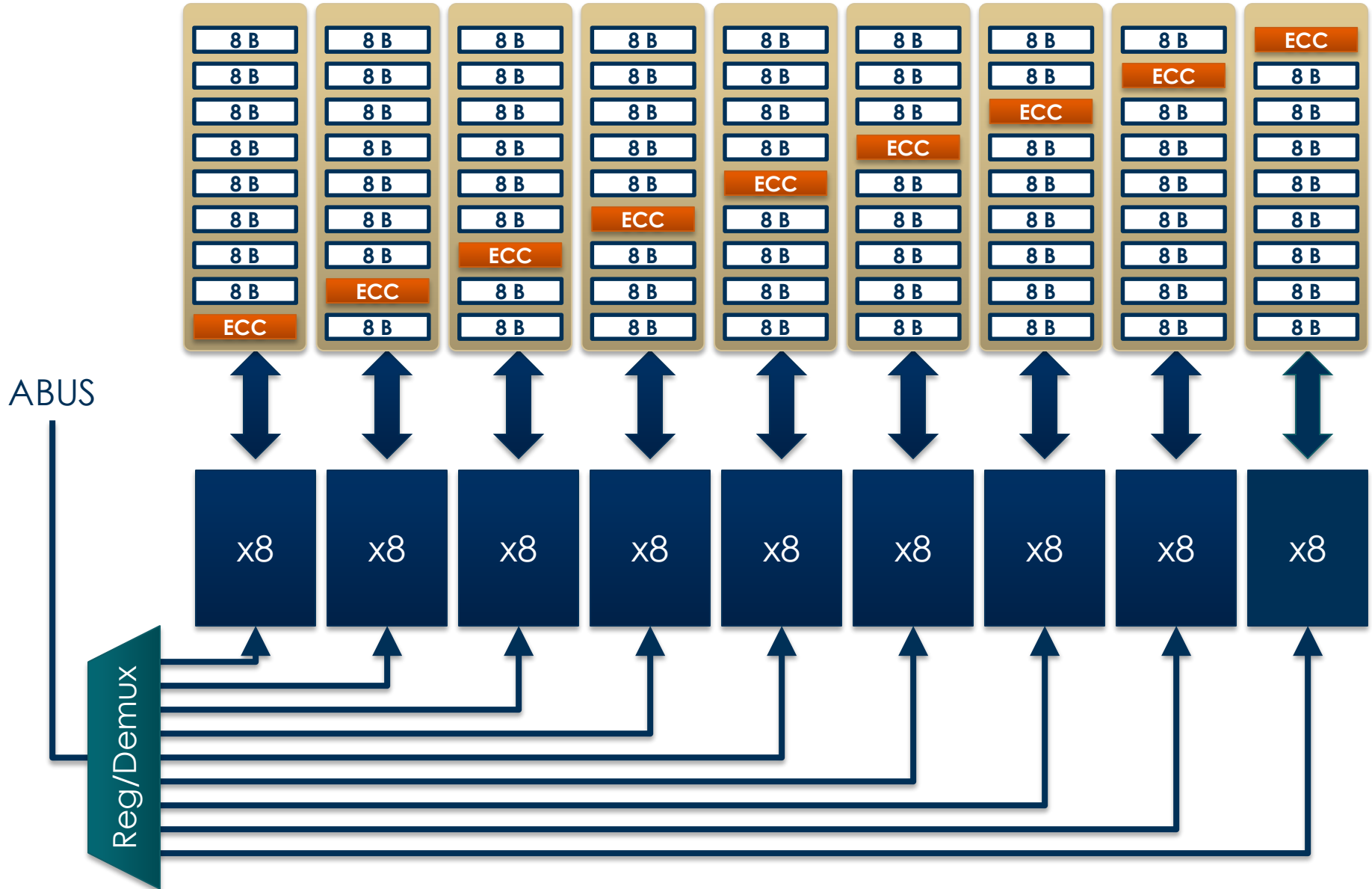
Short cache line

Sector cache

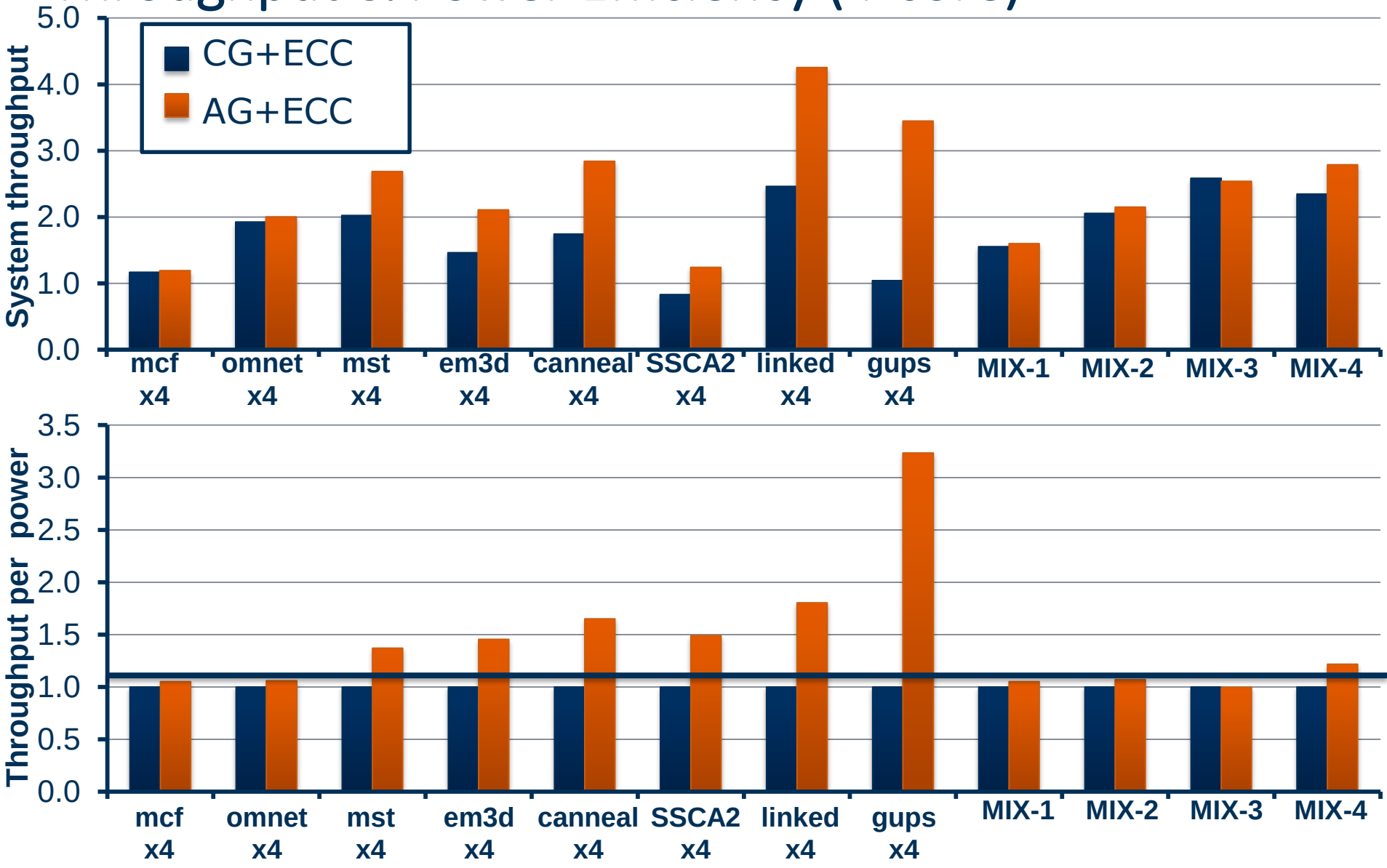
Common data layout for CG and FG



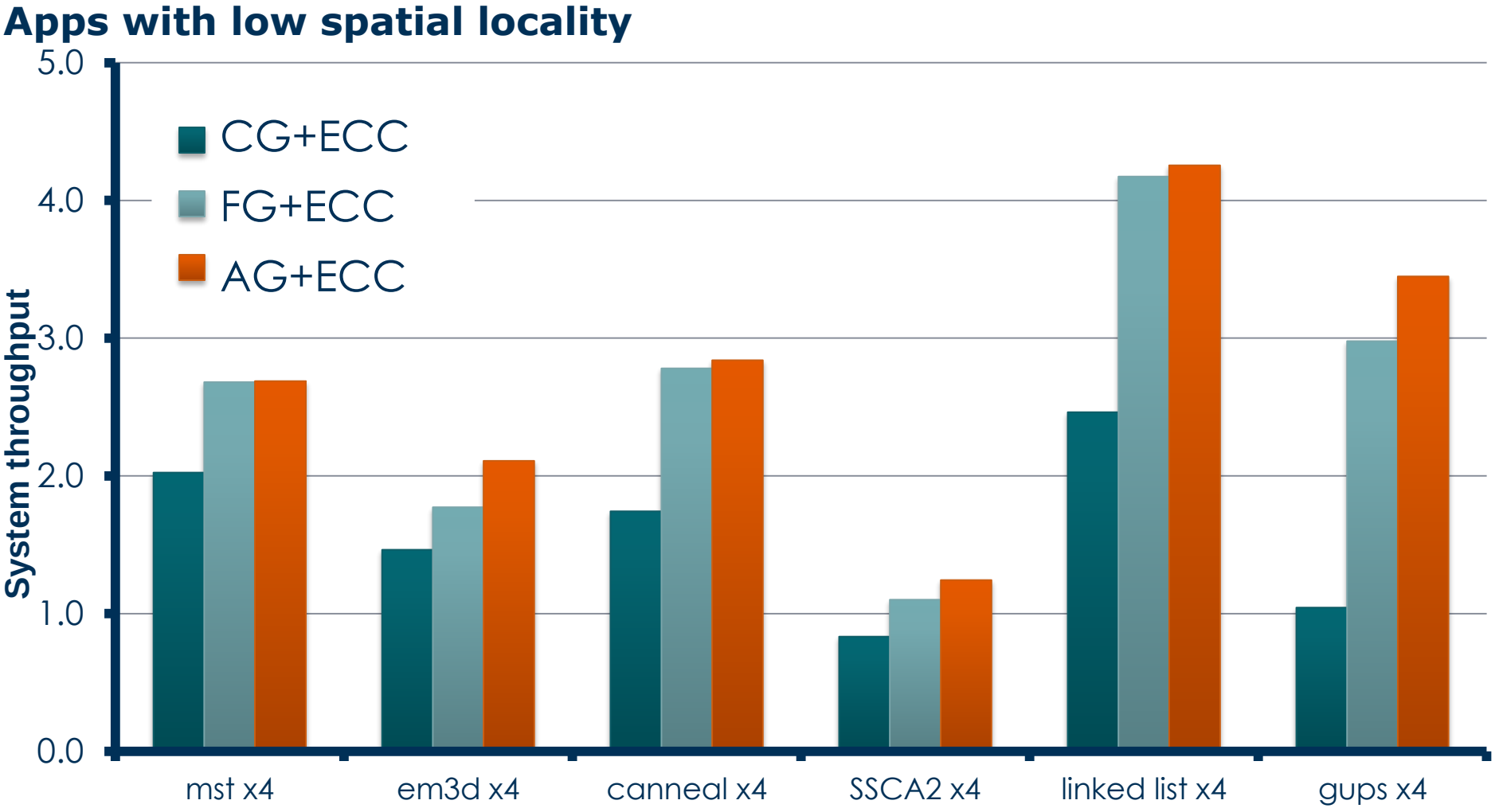
Common data layout for CG and FG



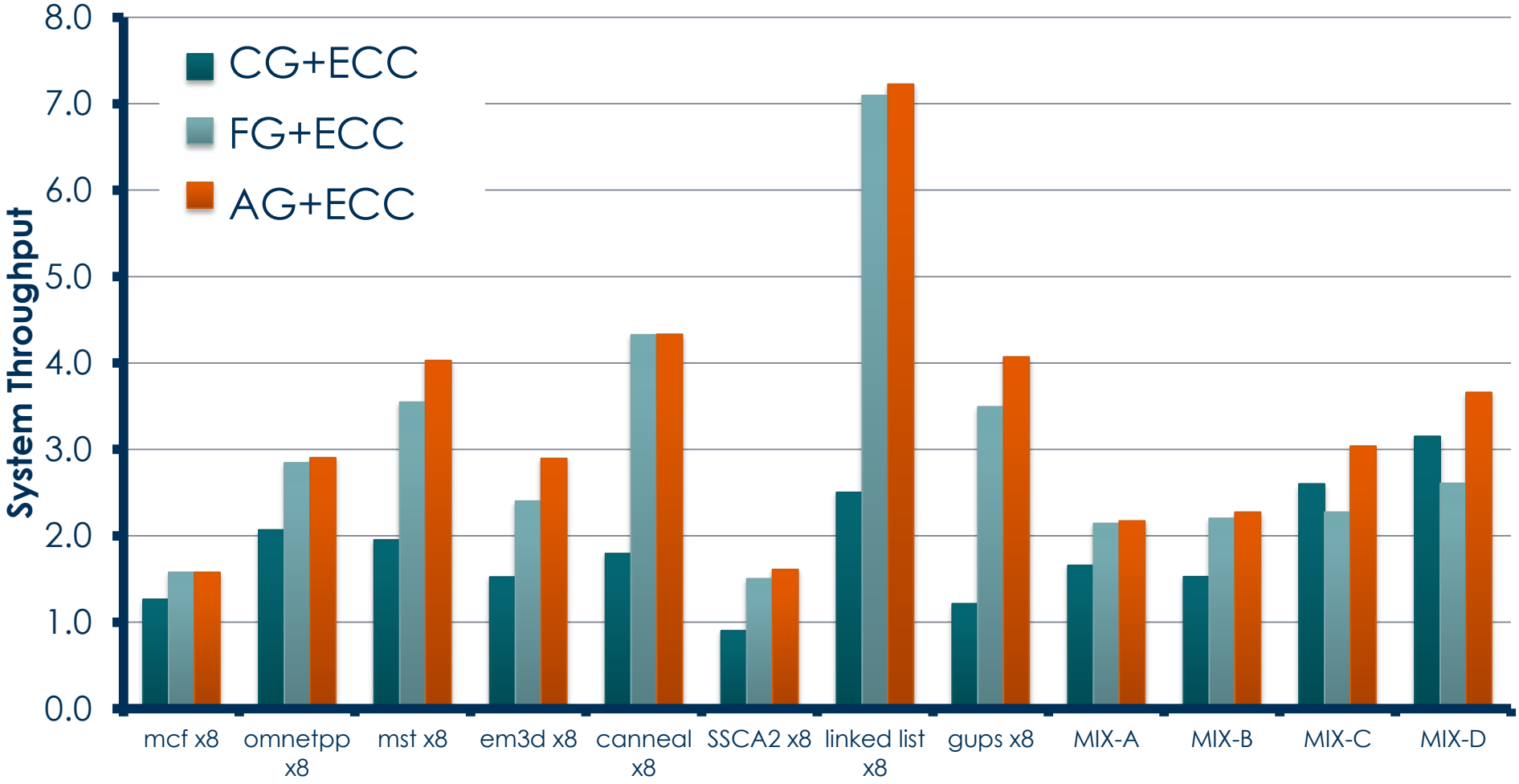
Throughput & Power Efficiency (4-core)



4-core results



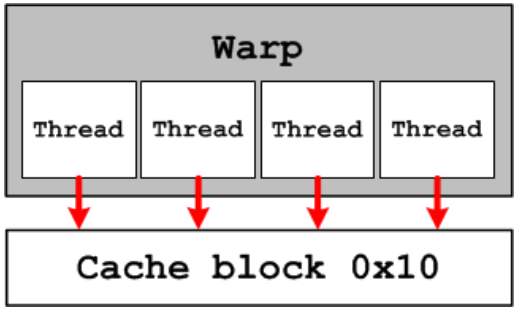
8-core results



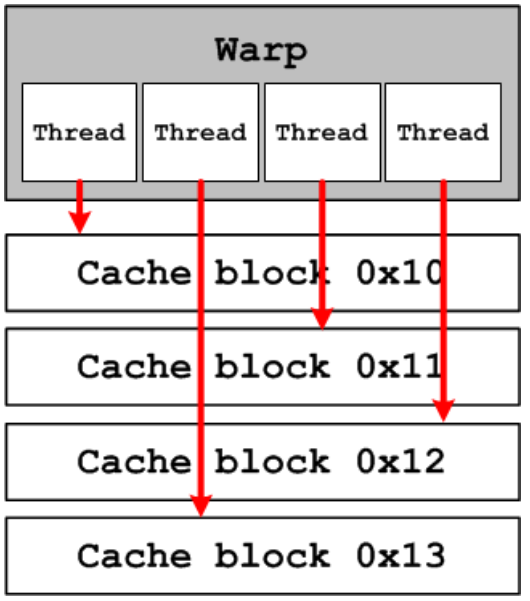
MIX-A: mcf x4, omnetpp x4 MIX-B: SSCA2 x2, mcf x2, omnetpp x2, mst x2	MIX-C: SSCA, mcf, omnetpp, mst, astar, hmmer, lbm, bzip2 MIX-D:libquantum x2, hmmer x2, mst x2, mcf x2

Irregularity even worse impact on a GPU

- Well-structured memory accesses are *coalesced* into a *single* memory transaction
  - e.g., all the threads in a warp access the ***same*** cache block
- A single warp (32 threads) can generate upto *32* memory transactions
  - e.g., threads in a warp access ***distinct*** cache blocks
- Irregularity severely degrades efficiency
  - Cache thrashing
  - FG memory access



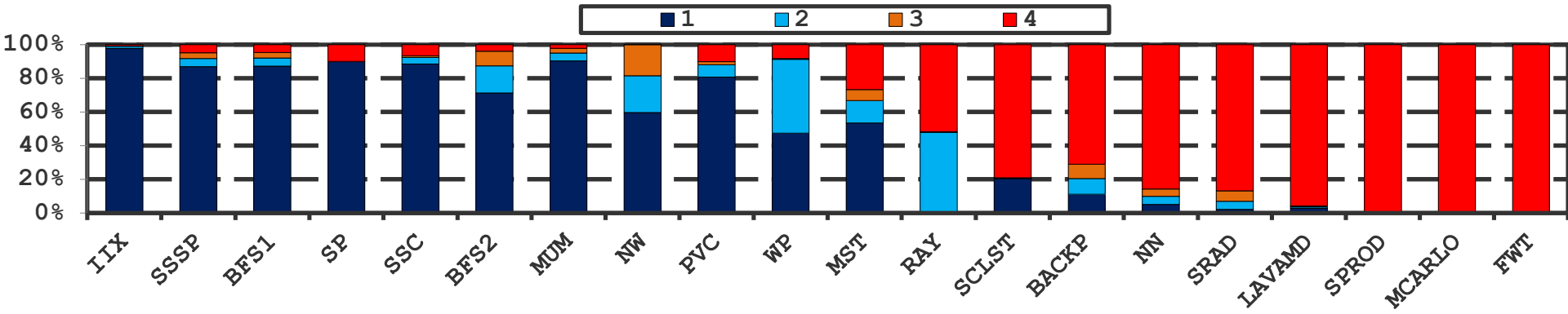
(a) Regular memory accesses



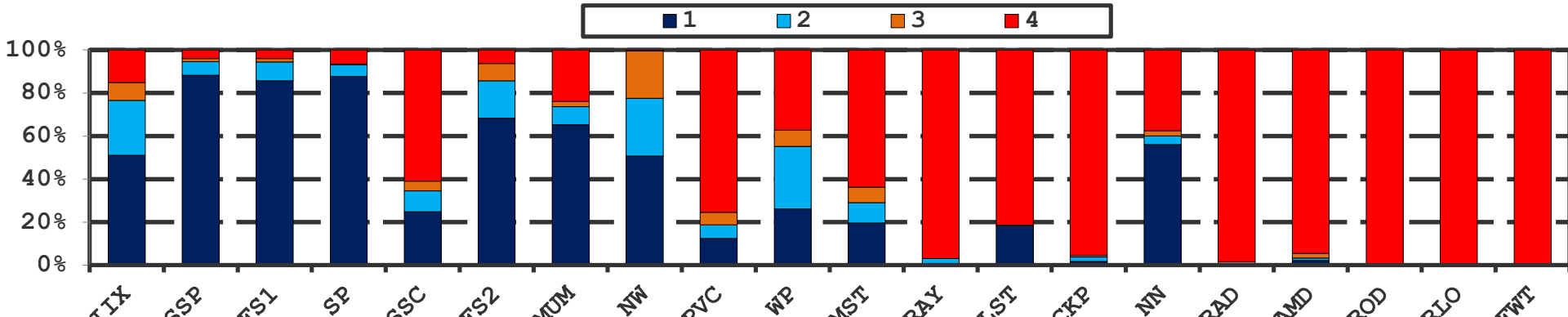
(b) Irregular memory accesses

Cache Block Locality (Spatial)

- Number of sectors referenced in L1/L2 cache blocks (CG-only memory-hierarchy, 128-Byte cache block)



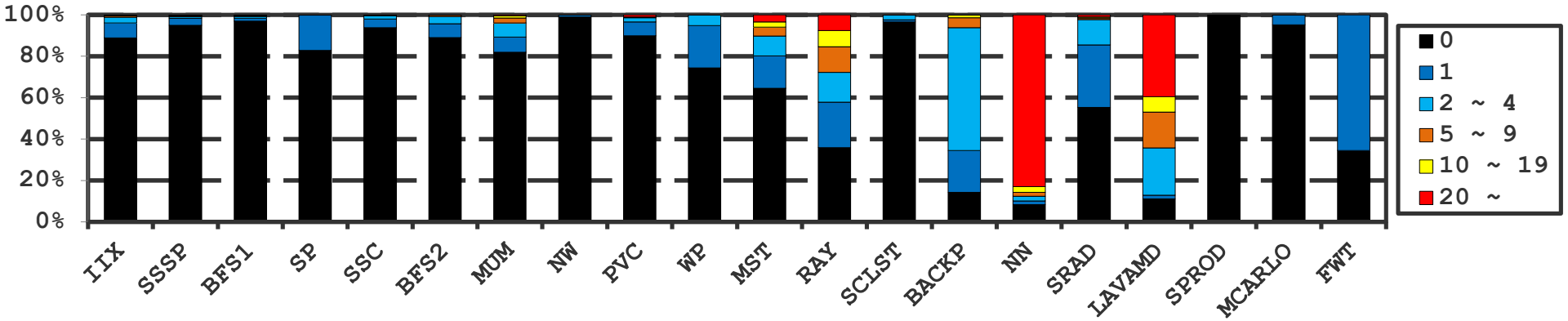
(a) L1 cache



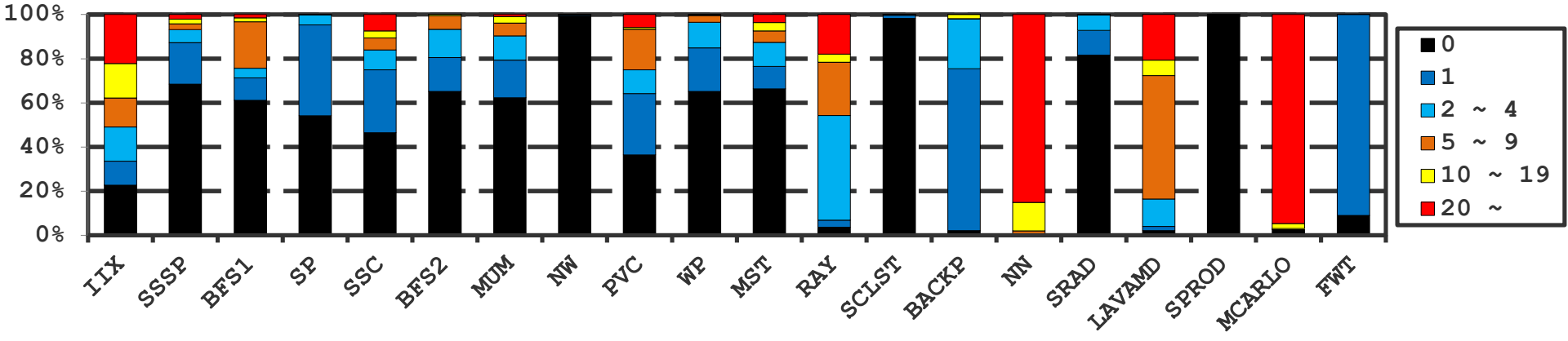
(b) L2 cache

Cache Block Locality (Temporal)

- Number of repeated accesses to L1/L2 cache blocks, after fill (CG-only memory-hierarchy)

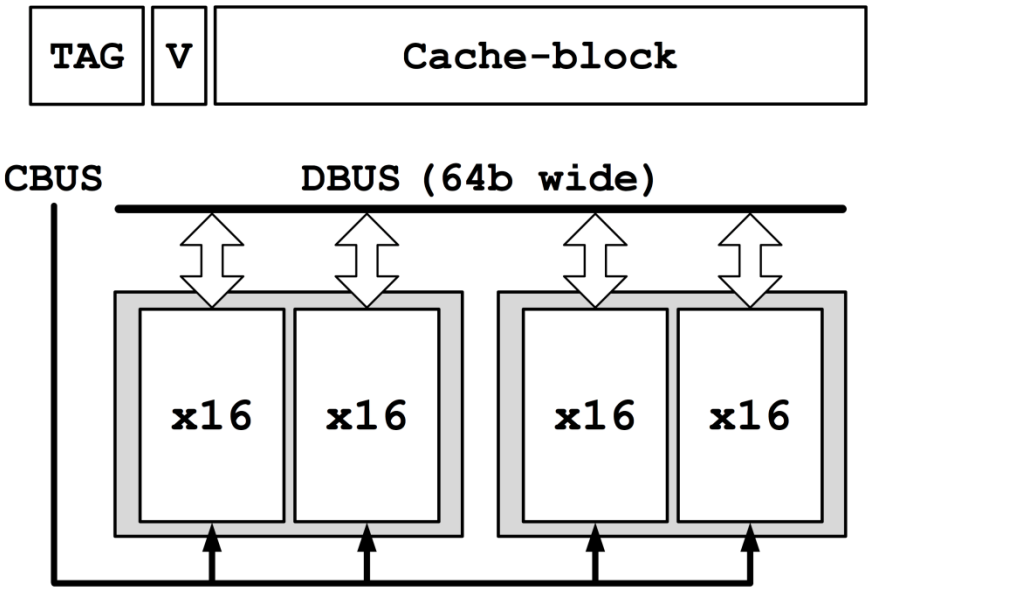


(a) L1 cache



(b) L2 cache

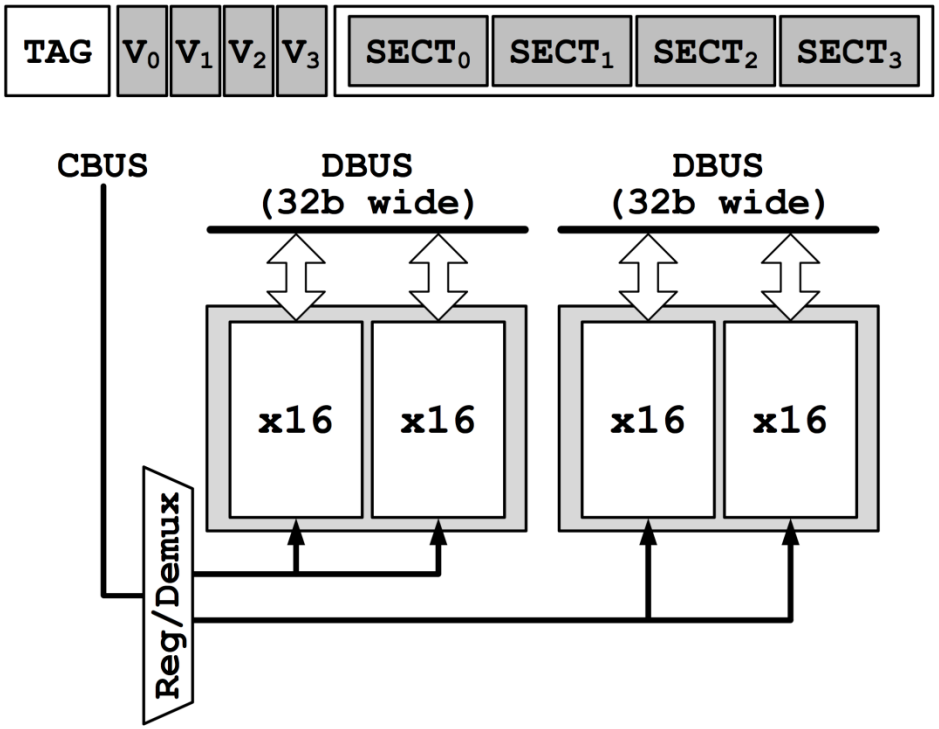
Memory Hierarchy (CG-only)



(a) Baseline cache and memory system (V: valid)

- Fermi (GF110) / Keplar (GK110) / Southern-Island
  - Each channel width: 64b
  - 4 ~ 6 channels overall, depending on product variants
- 64B (64b x 8-bursts) minimum access granularity

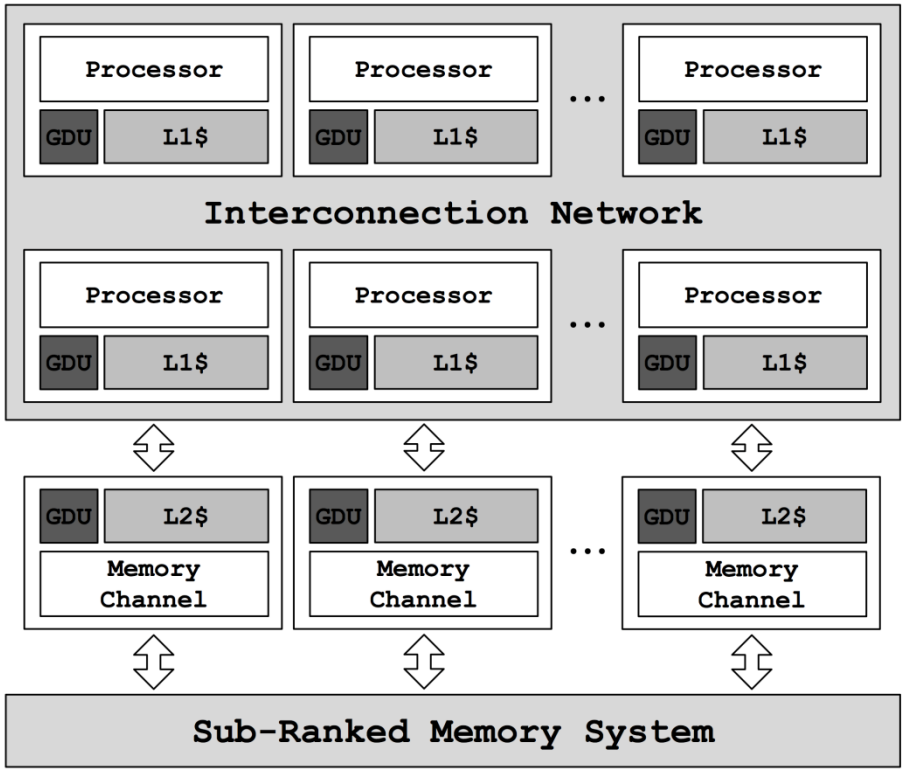
Memory Hierarchy (FG-enabled)



(a) Memory hierarchy with a sectored cache and a sub-ranked memory system (128B cache block, V: valid)

- Each channel divided into ***two*** sub-ranks
  - 32B (32b x 8-bursts) minimum access granularity
  - Allows finer control of data fetches from DRAM (64B vs 32B)

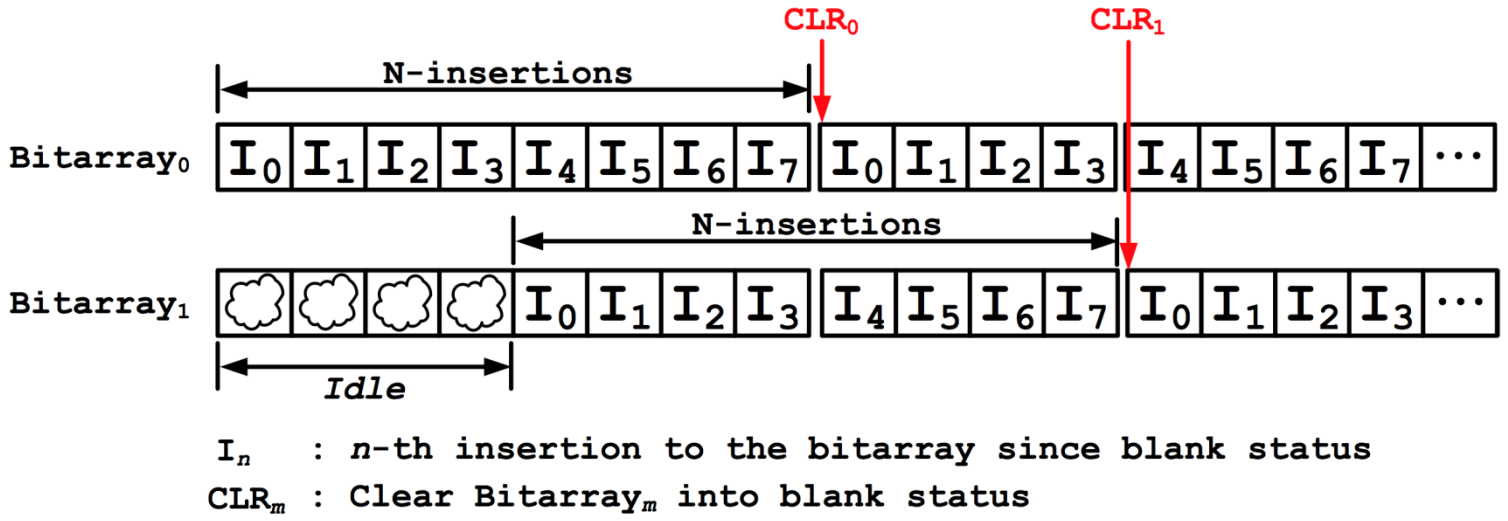
LAMAR: Locality-Aware Memory Hierarchy



GDU: Granularity Decision Unit

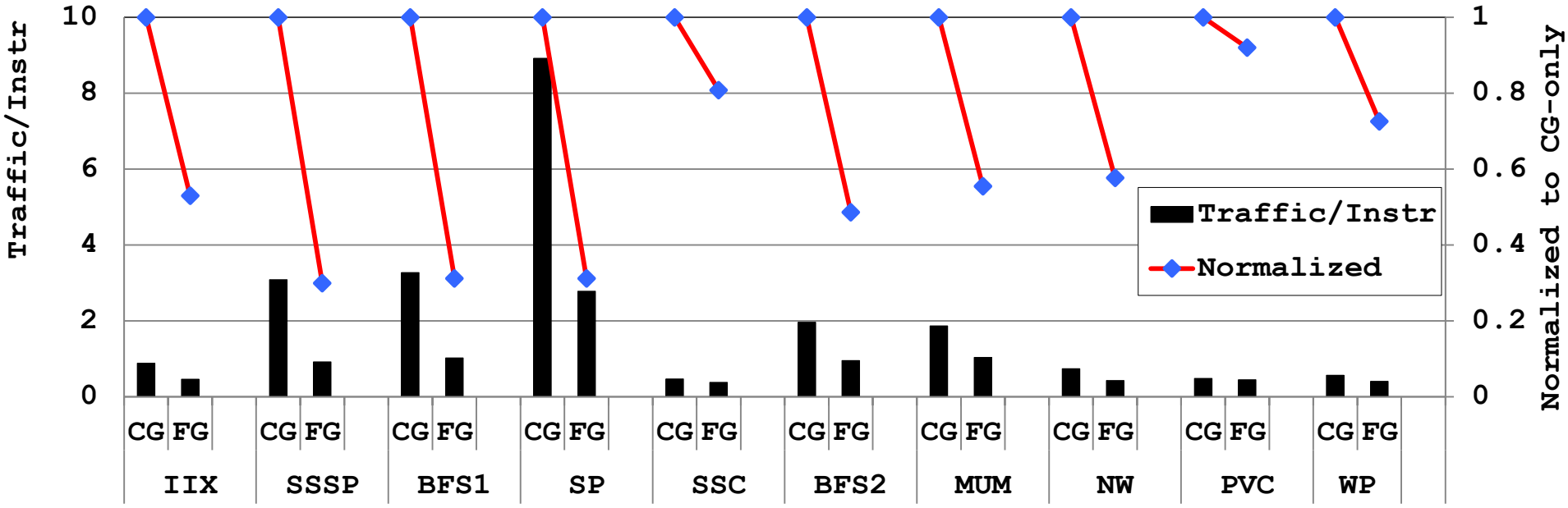
- Sectored caches + sub-ranked memory system
  - *Motivation*: massive multithreading + memory divergence limits cache block lifetime
    - Prefetching effectiveness ↓

Bi-modal Predictor (*all* or *on-demand*) as GDU



- GPUs contain 10s / 100s number of cores
  - *Spatial-pattern predictor* is heavy-weight / not scalable
- *Dual*-bitarray bloom-filter
  - Light-weight, ***temporally overlapped*** for history preservation
  - Always maintain *subset* of insertion-history

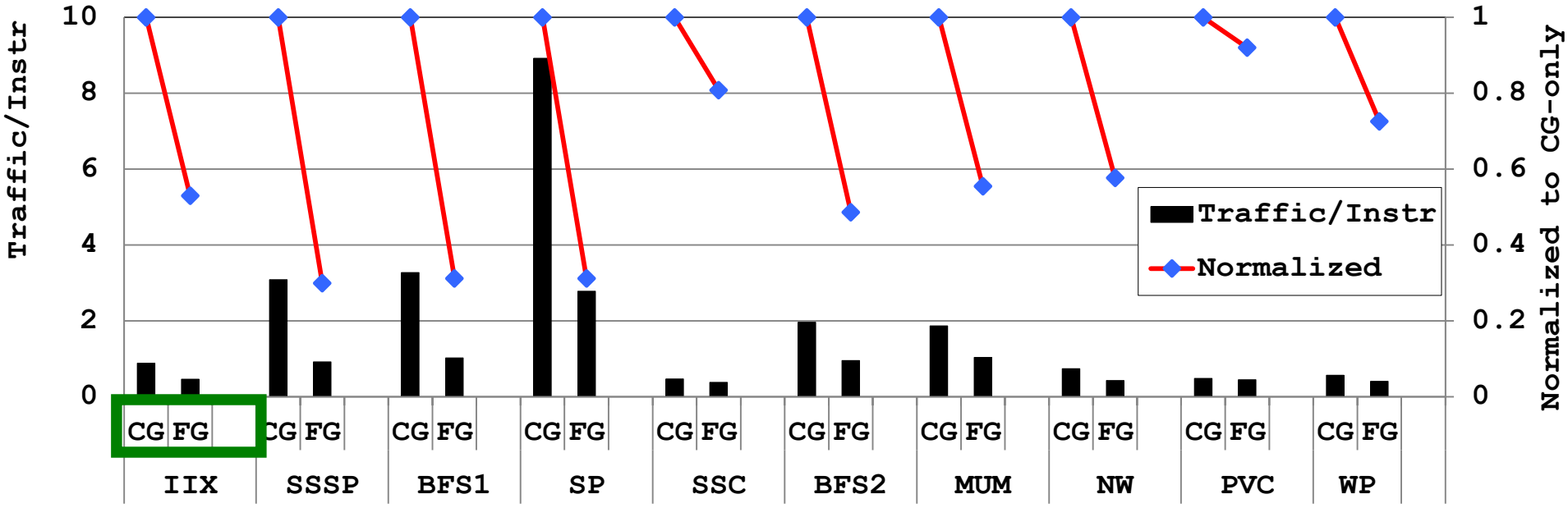
Off-chip byte traffic (normalized to # of instructions)



Lower is better

Applications: highly divergent ones which can benefit with FG-accesses

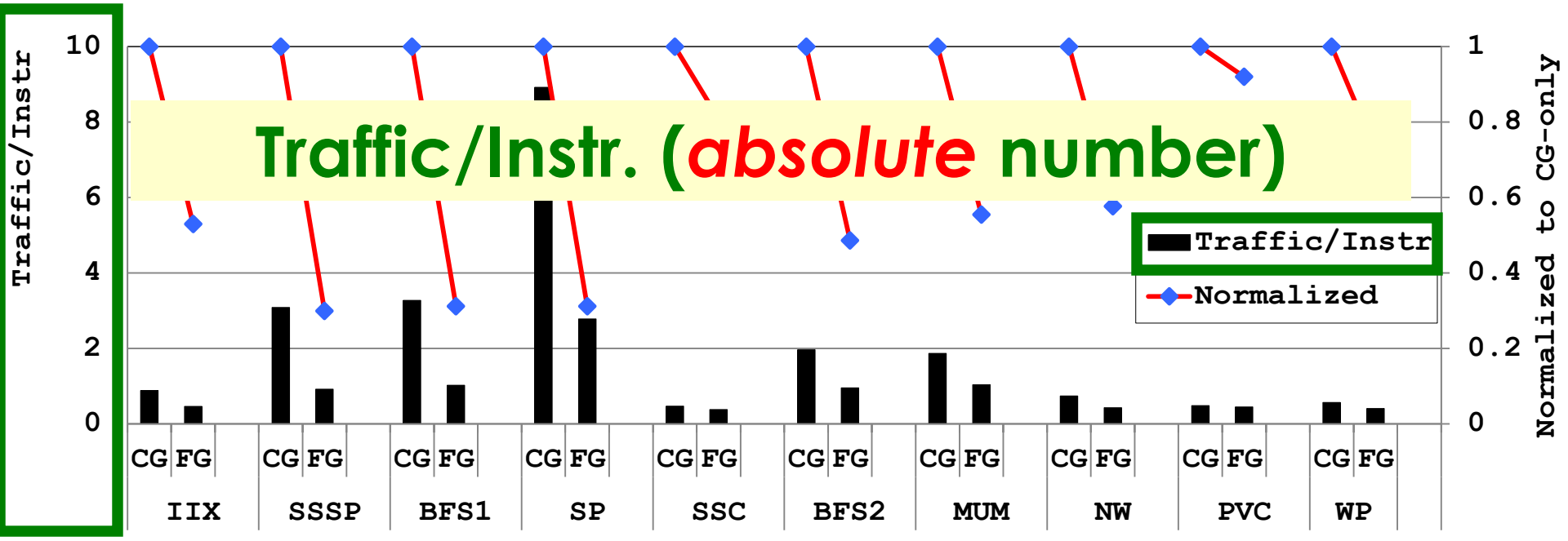
Off-chip byte traffic (normalized to # of instructions)



CG : *Static-GDU with CG*-only fetches (Baseline)

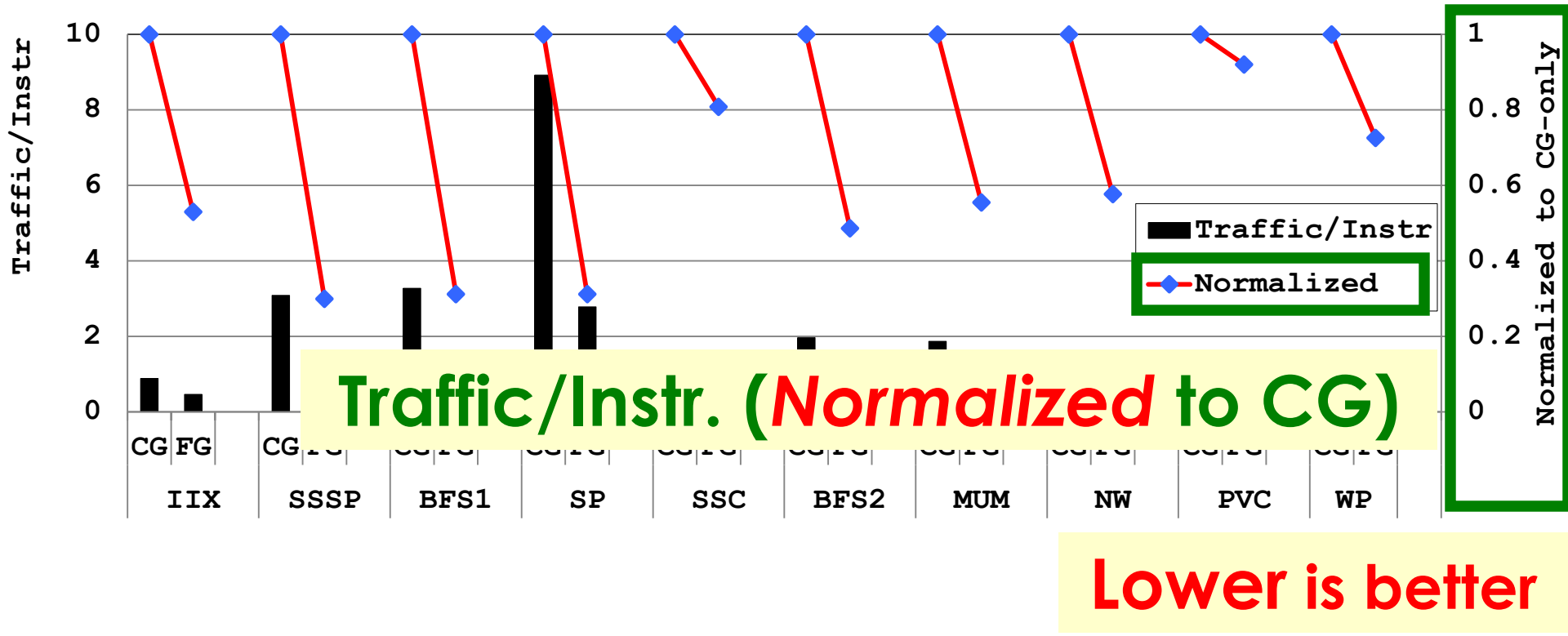
FG : *Static-GDU* with *FG*-only fetches

Off-chip byte traffic (normalized to # of instructions)



Lower is better

Off-chip byte traffic (normalized to # of instructions)

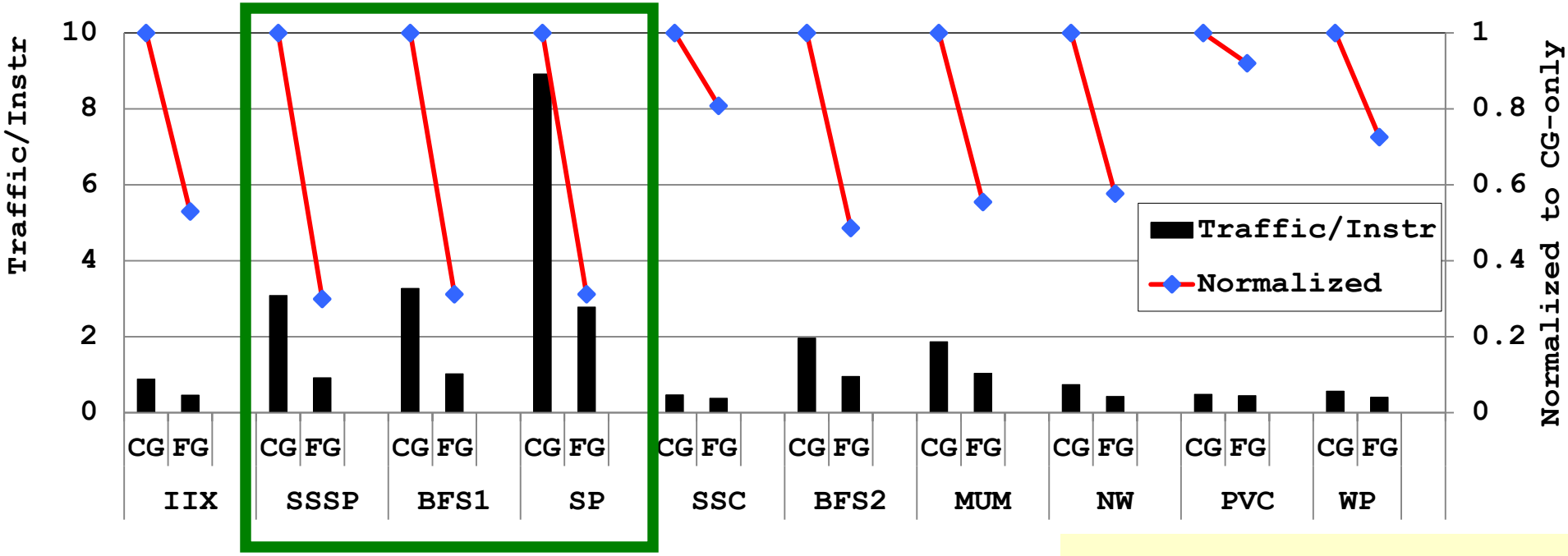


Traffic/Instr. (*Normalized* to CG)

Lower is better

Off-chip byte traffic (normalized to # of instructions)

FG: avg. 53% (max 71%) ↓

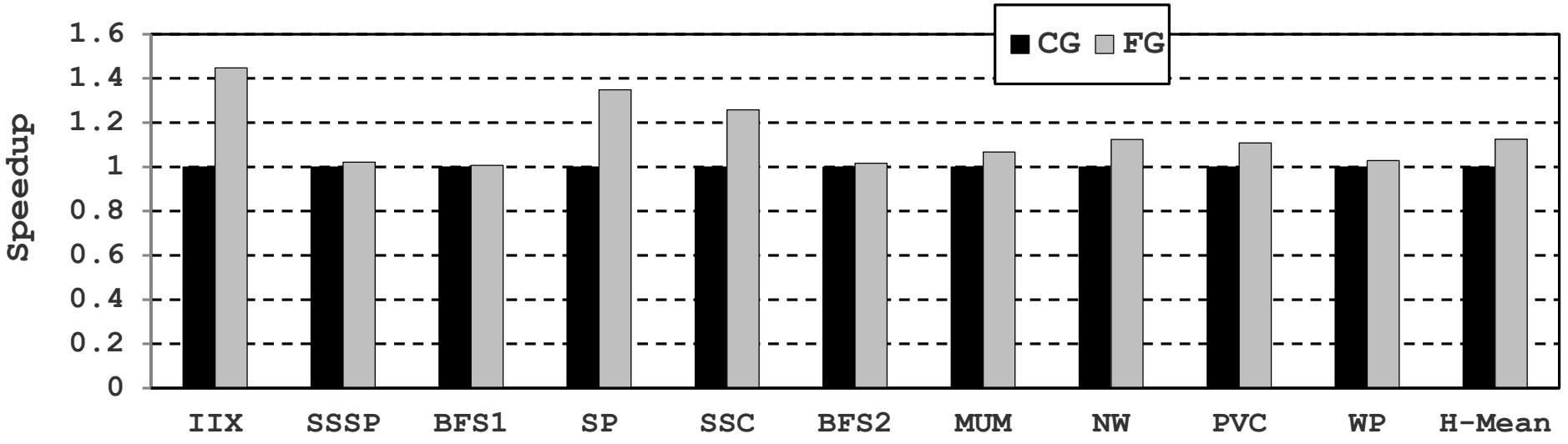


Lower is better

FG-fetches reduce number of READ and WRITE transactions through memory channels.

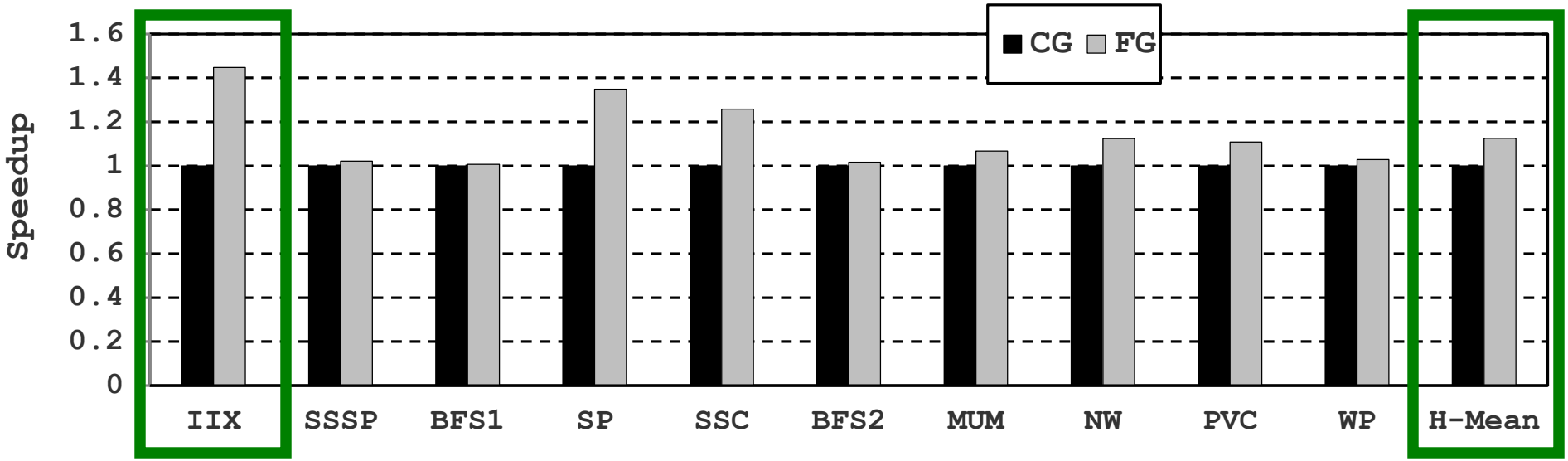
Performance improvement

Higher is better



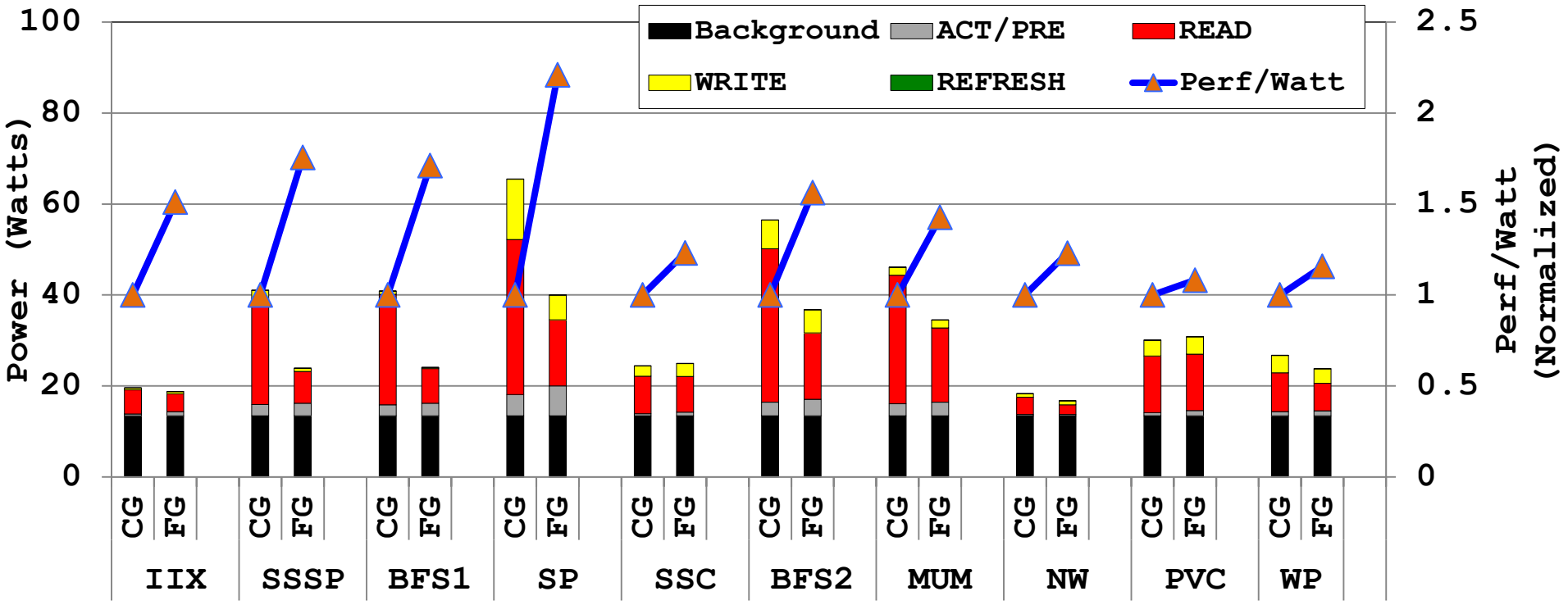
Performance improvement

Higher is better

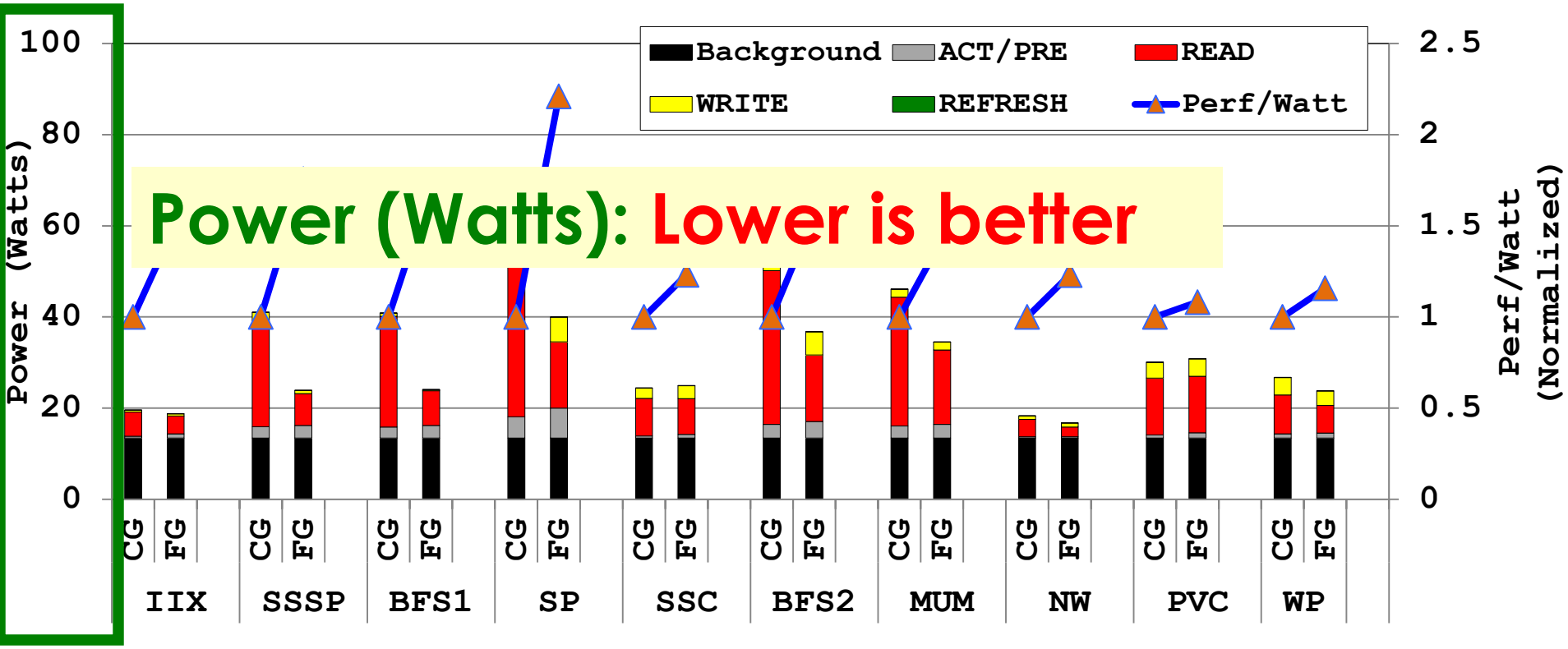


FG: avg. 13% (max 47%) ↑

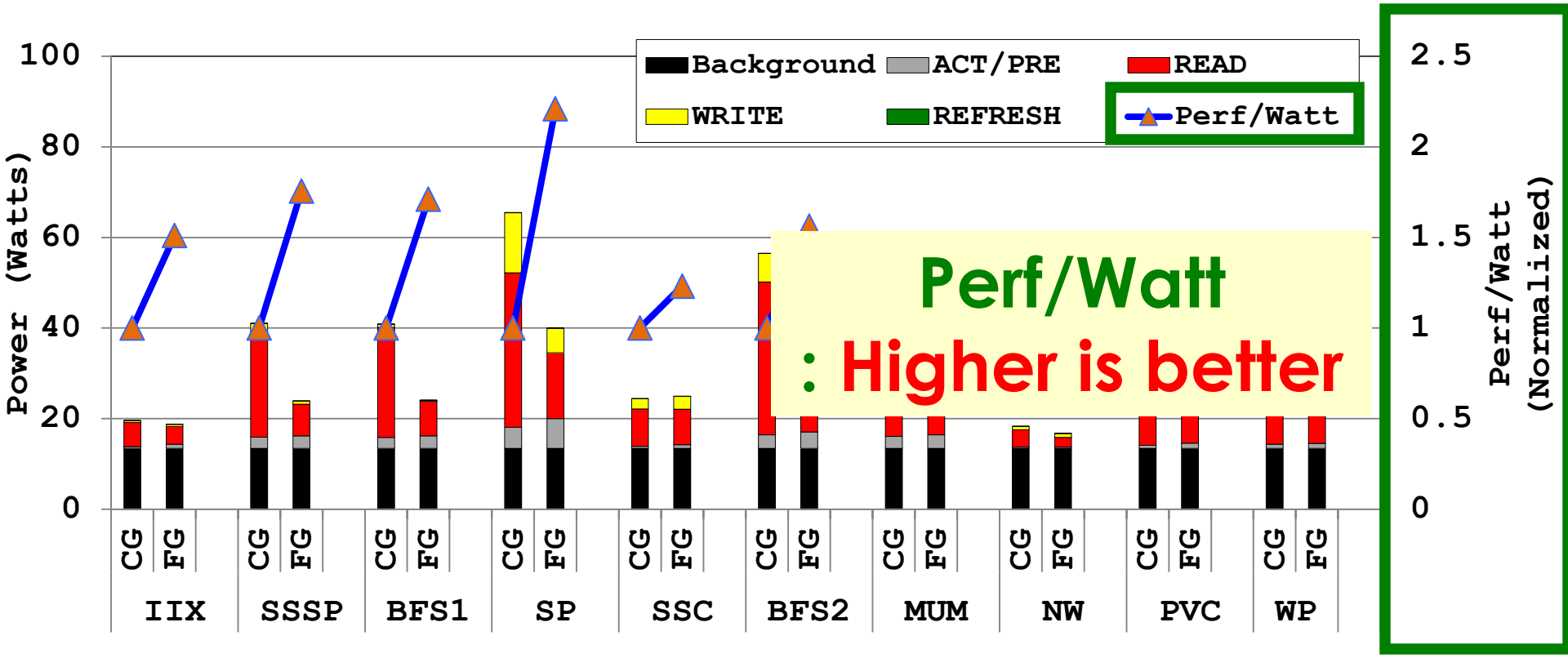
DRAM power consumption



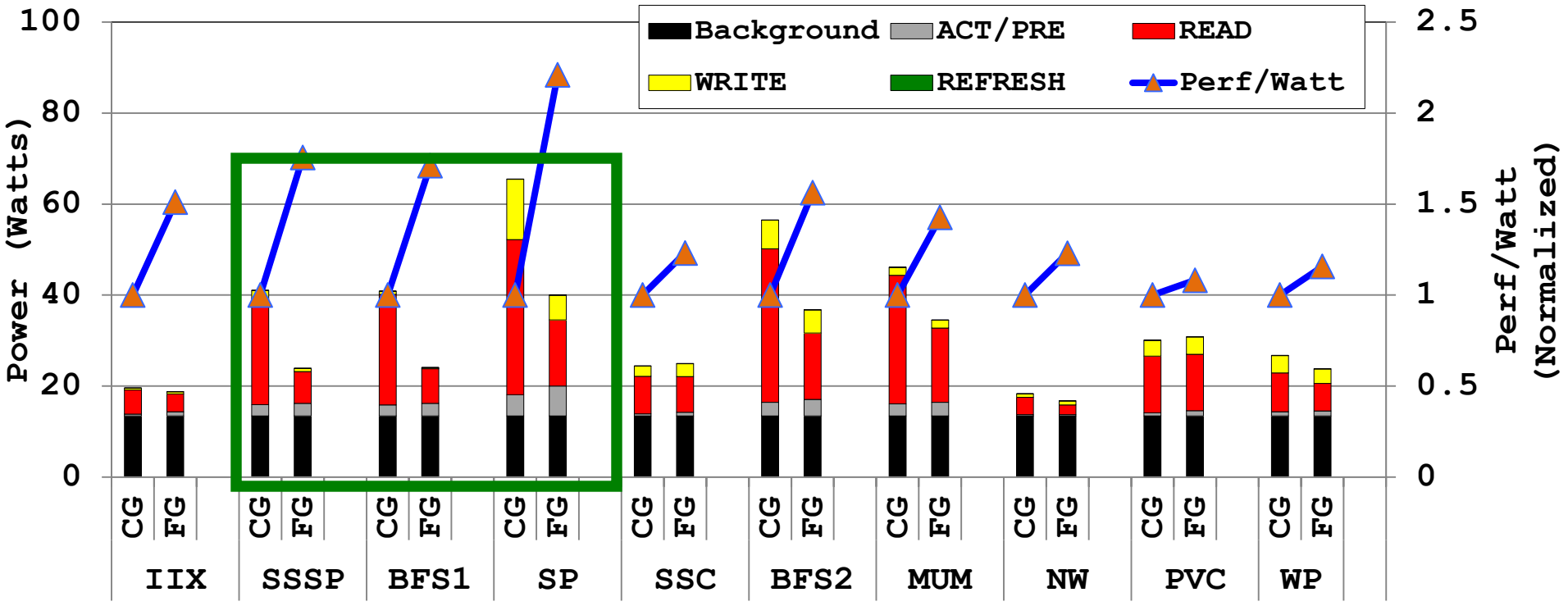
DRAM power consumption



DRAM power consumption



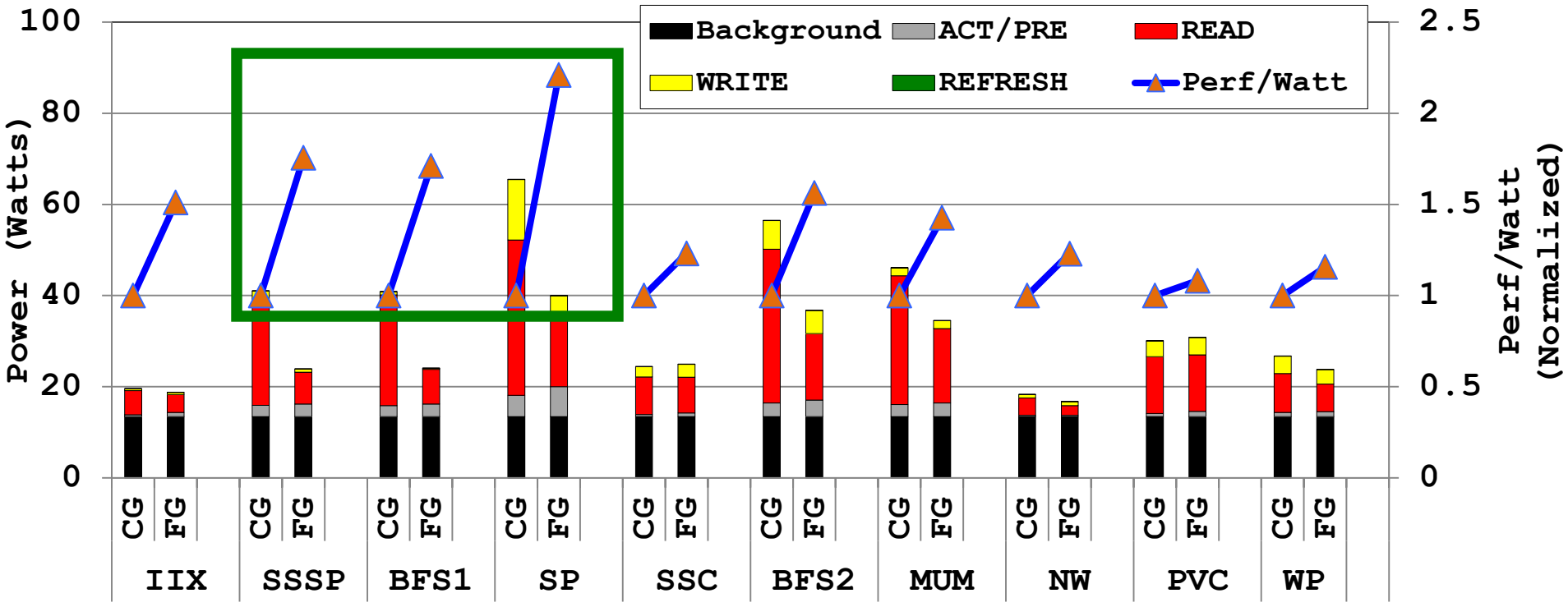
DRAM power consumption



< DRAM power >

FG: avg. 19% (max 42%) ↓

DRAM power consumption



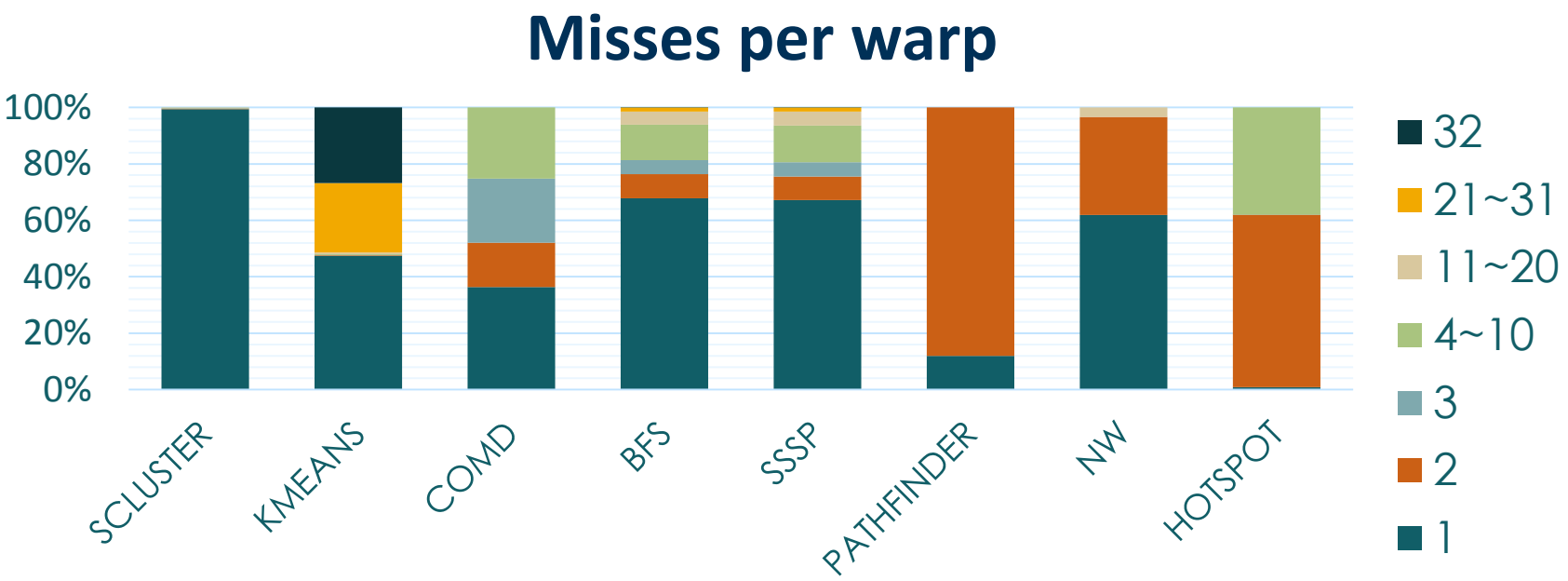
< Perf/Watt >

FG: avg. 42 % (max 121%) ↑

ROBUST GPU CACHING

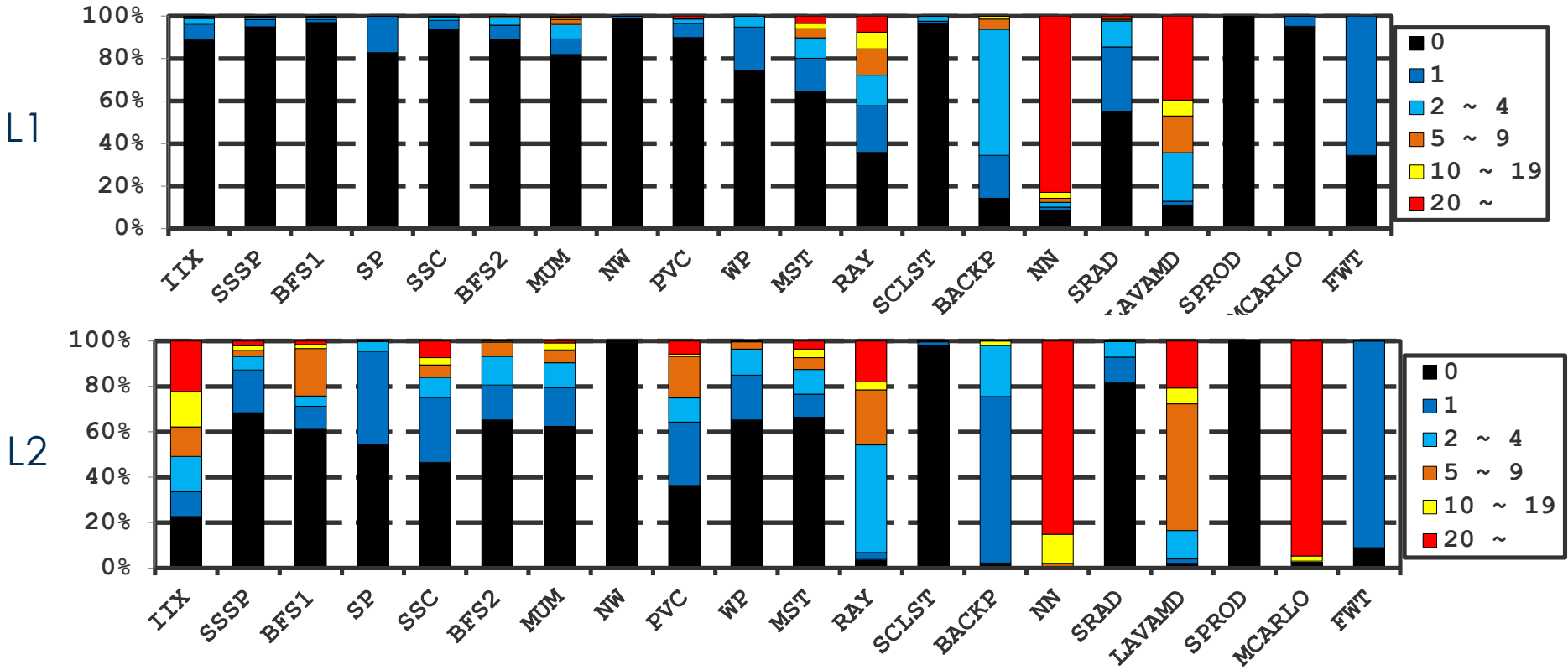
Memory divergence and caching don't mix well

- Massive multithreading + irregular memory accesses
  - Bursts of concurrent cache accesses within a given time frame
- Cache capacity per thread very small
  - Only 24B/thread
  - Xeon has 16KB/thread!



Memory divergence and caching don't mix well

- Massive multithreading + irregular memory accesses
  - Bursts of concurrent cache accesses within a given time frame
- Cache capacity per thread very small
  - Most cache lines fetched and evicted before reused

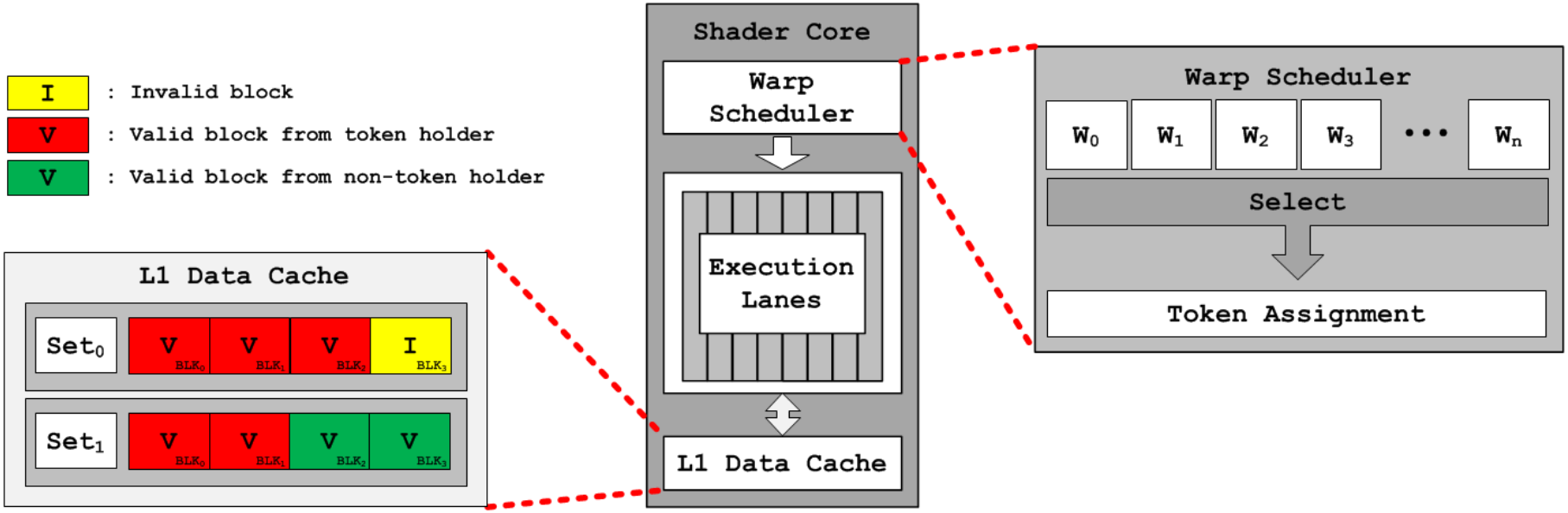


Possible **solution: throttle parallelism**

- Throttle number of schedulable threads
  - Fewer threads → less thrashing
    - Rogers et al., MICRO 2012
    - Kayiran et al., PACT 2013
- But, sacrifice latency hiding
  - Not robust
- No better than software tuning

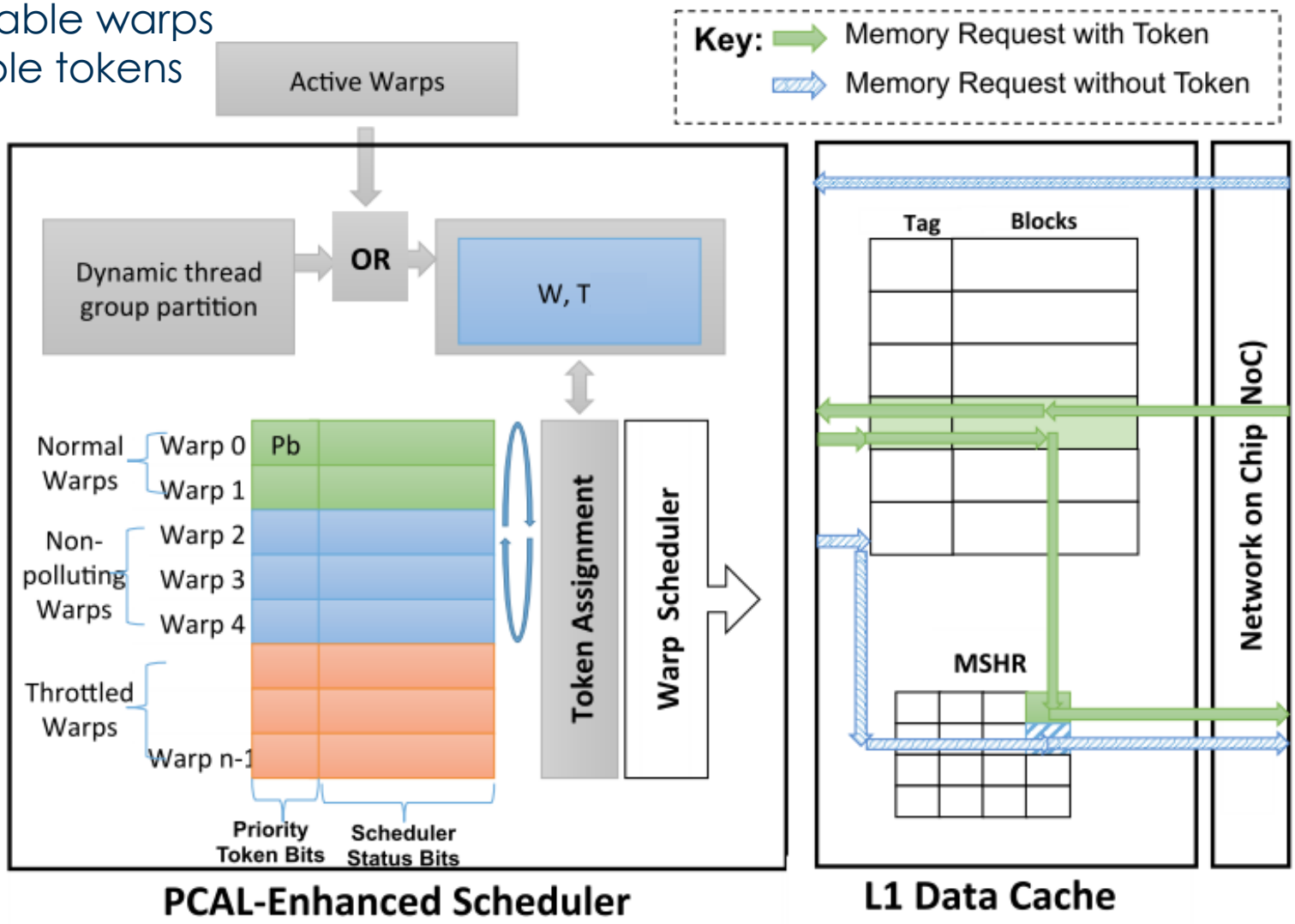
Better solution: Priority-based Cache ALlocation

- Bypass the cache instead of throttling parallelism
- Prioritize some warps for cache allocation/deallocation
  - Oldest ***T*** warps get cache-use ***token***
- Other warps use cache opportunistically



PCAL hierarchy

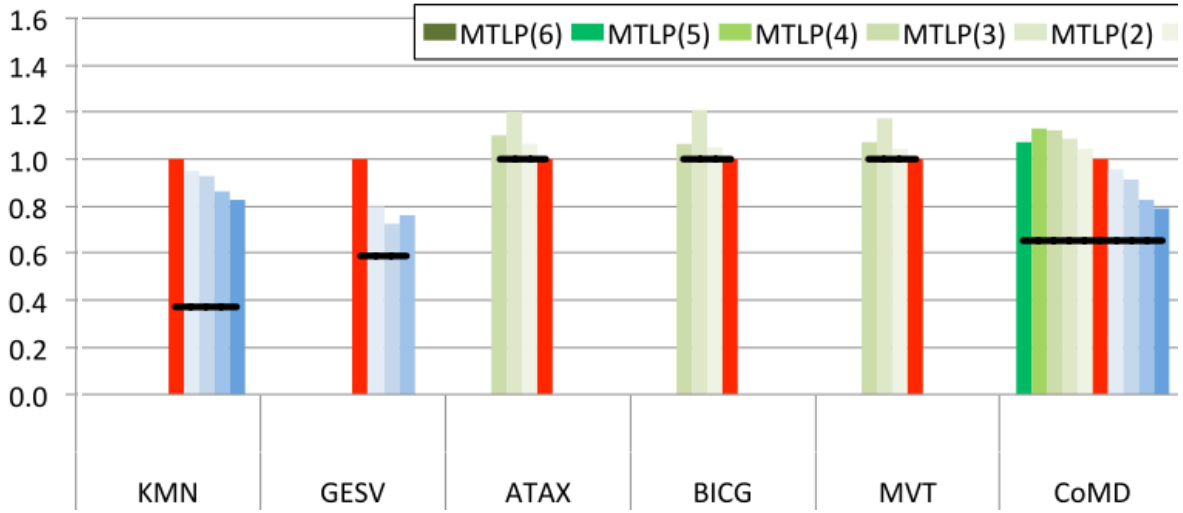
W: schedulable warps
T: # available tokens

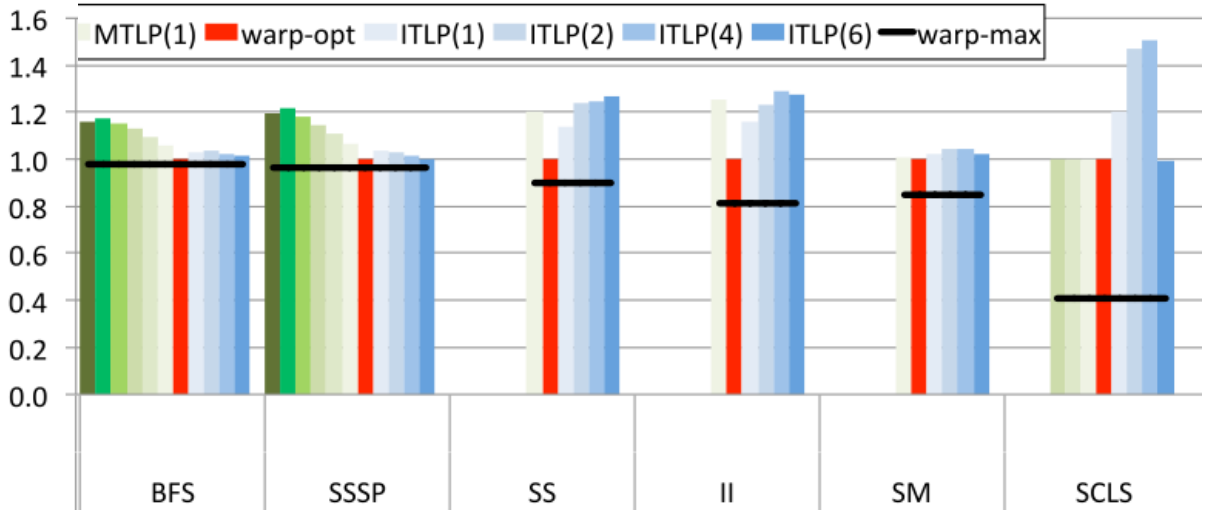


PCAL is robust and performant

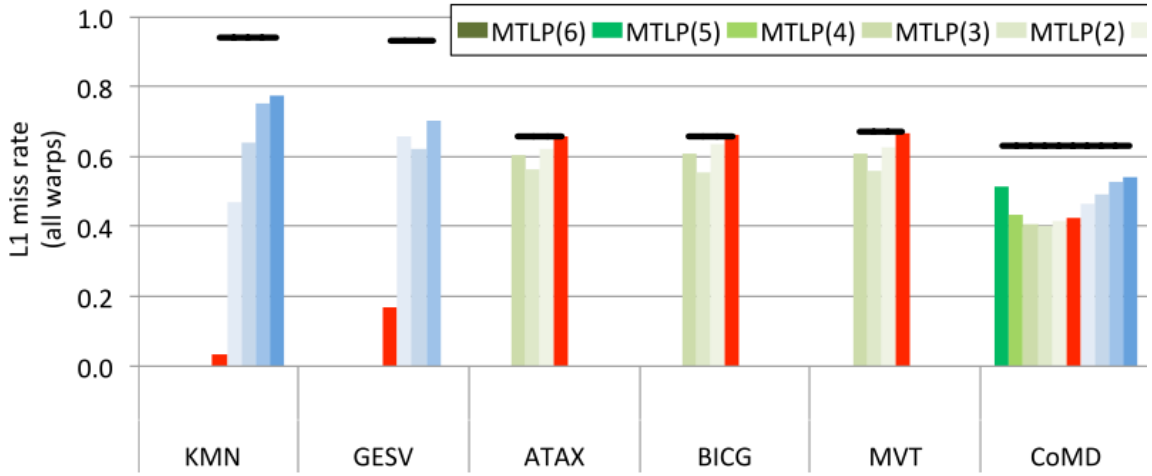
- Two operating points:
  - ITLP: increase TLP while maintaining hit rate
    - T = Wopt, W > Wopt
  - MTLP: maintain TLP and improve hit rate
    - T > Wopt, W = Wopt

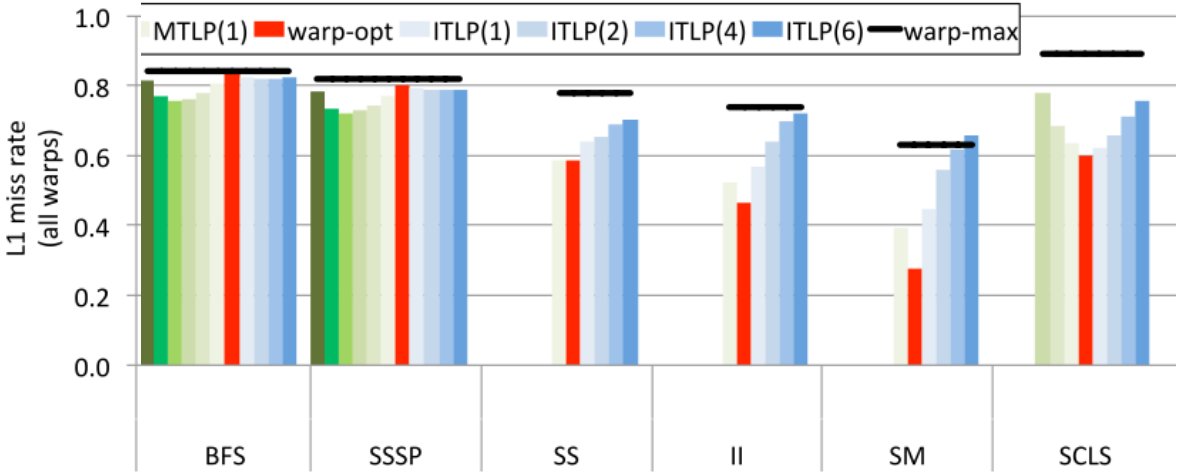
Performance improvement (relative to throttling)



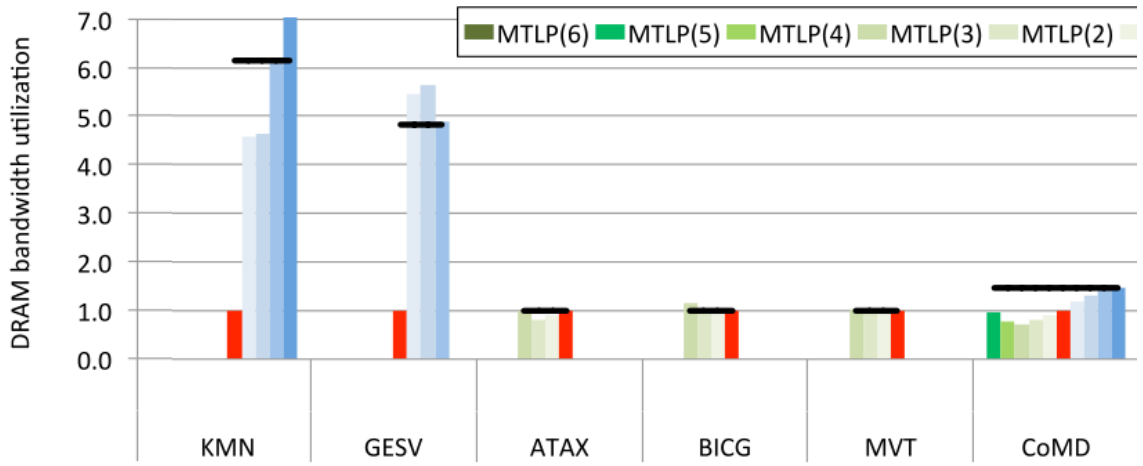


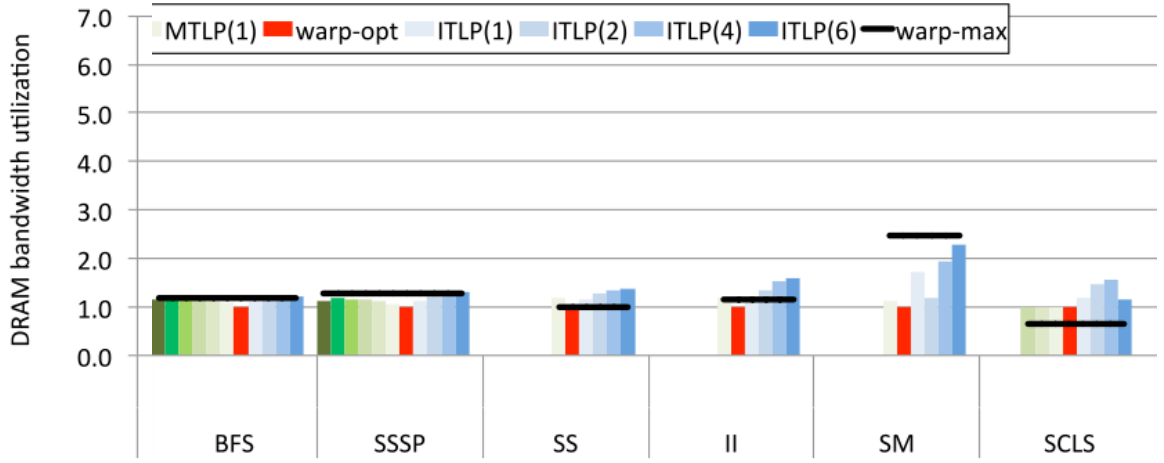
L1 miss rate improvements





Off-chip throughput improvement (rel. to throttling)





Also working on …

- DGMS with chipkill
- LAMAR with ECC (and chipkill)
- Addressing memory divergence better on GPUs

- And:

Resilience at scale

- Containment domains
  - Elevate resilience to first-class abstraction
  - Proportional, hierarchical, distributed, and cross-layer
  - Programming model, runtime, compiler, and hardware
- Avoiding SDCs with proportional (low) overhead
  - Approximate duplication
  - Circuit-level detection
  - Algorithm hints and compiler optimizations
- Proportional memory protection
  - Adapt and tune protection
    - Meet diverse and dynamic application and system requirements
  - Virtualized and multi-tier ECC, other adaptations

Resilience for emerging technologies

- Proportional NVM wearout-protection
  - Fine-grained NVM wearout-protection
  - ECC schemes for MLC memories
- On-package memory protection
  - Reliability a huge concern (replacement costs)
  - Limited capacity and granularity concerns
  - Rich design space
- Parallel pipelines with high process variation
  - Effective SIMD timing speculation in near-threshold
- Energy-efficient STT-RAM (w/ Orshansky and Samsung)
  - Dynamic write architecture for high-reliability writes
  - Array islands and adaptive placement

Proportional and adaptive memory

- Resilience schemes
- Adaptive and dynamic access granularity
  - Efficiency of coarse-grained access for spatial locality
  - Performance of fine-grained for highly irregular access
  - Power savings with sub-row DRAM activation
- Reducing interference and improving locality
  - Bank-partitioning eliminates inter-core row conflicts
  - Constructively-interleaved page allocation creates locality in GPU context
- Adaptive QoS for heterogeneous processors
  - Balancing QoS for real-time and best-effort cores
  - Consistent and cooperative QoS prioritization

Up-and-coming platforms

- Improved GPU μarch for divergent control glow
  - Robust and effective thread compaction
  - Increasing parallelism with dual-path execution

- Cloud Radio Access Network (C-RAN)
  (w/ Heath, De Veciana, Evans, and Huawei)
  - Wireless base-stations are expensive and inefficient
  - Desire to co-process signals across base-stations
  - Move processing to a specialized cloud
    - Improved system and facilities
    - Improved maintenance
    - Enable joint processing

Conclusions

- Regularity good for hardware
- Irregular good for (lots of) software
- Dynamic Adaptivity:
  No-compromise ***robust*** architecture

- Support regularity as well as possible
- Introduce features for irregularity
- Can do a lot just at microarch level
  - But even more if involving other parts of the system