Near-Stream Computing: General and Transparent Near-Cache Acceleration

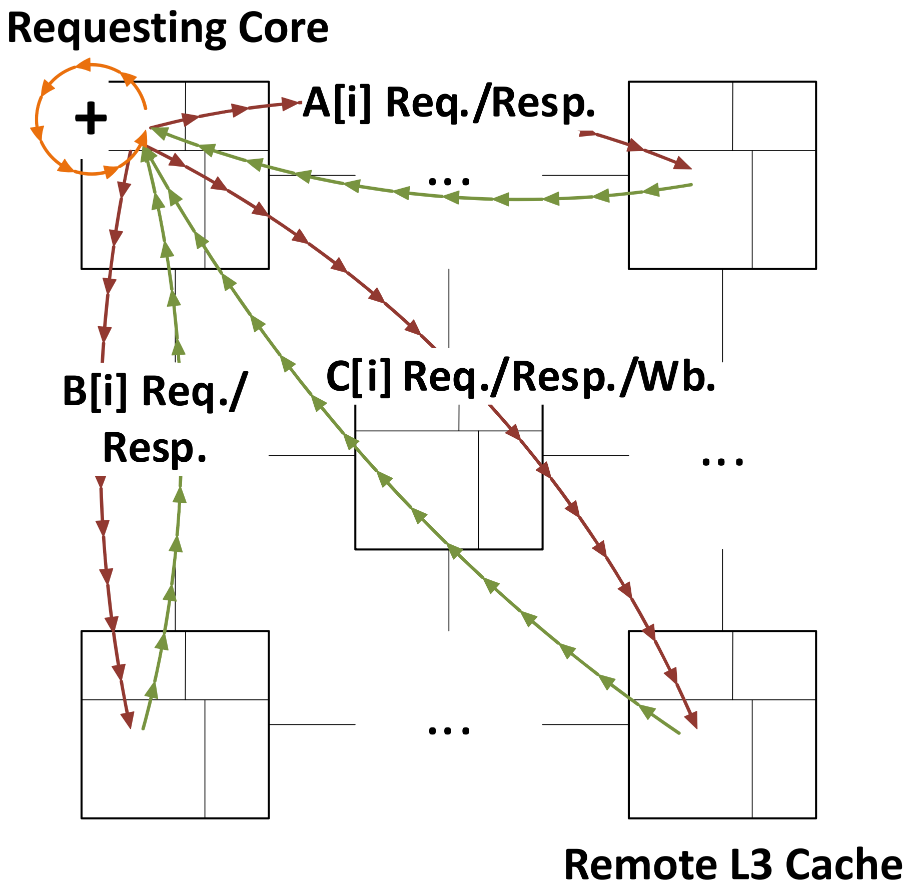
Zhengrong Wang, Jian Weng, Sihao Liu, Tony Nowatzki

UCLA

April. 2022

Near-Data Computing to Decentralize Computation

```
while (i < N)
  C[i] = A[i] + B[i]
  i++
```

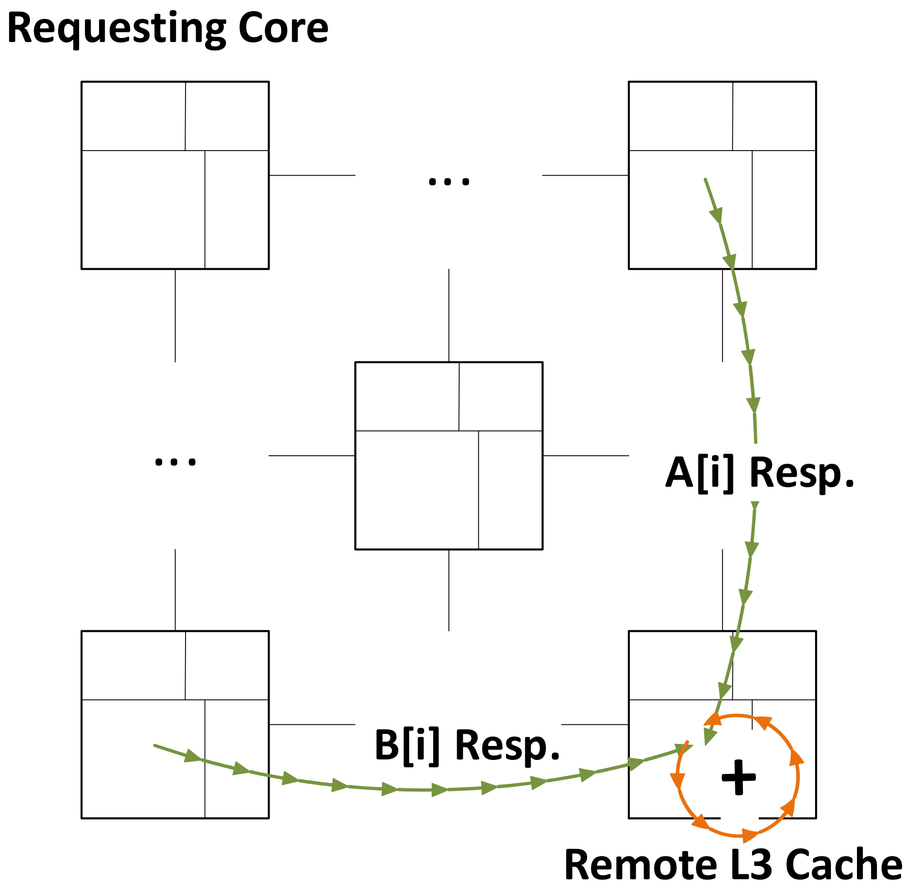


Data movement become increasingly the bottleneck.

- Computation centralized in the core.
- More expensive to fetch data as system scales up.
- Prefetching/streaming can only partially help.

Near-Data Computing to Decentralize Computation

```
while (i < N)
  C[i] = A[i] + B[i]
  i++
```



Data movement become increasingly the bottleneck.

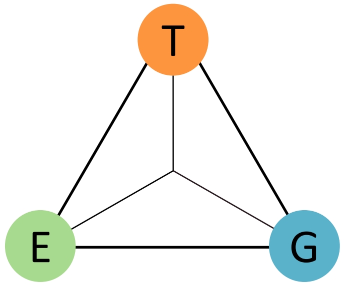
- Computation centralized in the core.
- More expensive to fetch data as system scales up.
- Prefetching/streaming can only partially help.

Near-data computing decentralizes the computation.

- Location: memory, storage, ***LLC (Our Focus)***, …
- Technology: in-situ, inorder cores, FPGA, …

Existing NDC works fall short to fully cover:

- Transparency.
- Synchronization-Efficiency.
- Generality.



… due to inappropriate *abstraction* choice:

- Instructions, user-defined functions, threads, …

Features of An Ideal NDC Abstraction

Transparency.

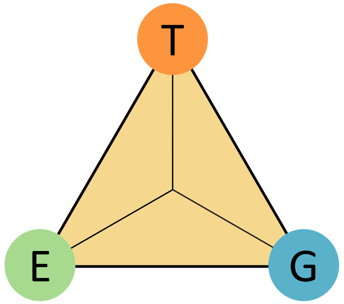
- No manual programming, auto-transformation, sequential semantics.

Synchronization-Efficiency.

- Low traffic (hops) to synchronize core and offloaded NDC, alias detection.

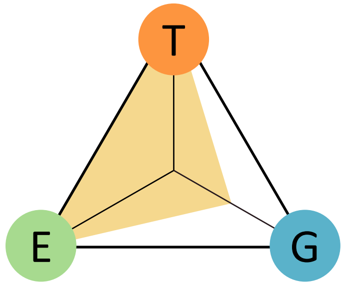
Generality.

- Arbitrary combination of memory access pattern and compute operation.



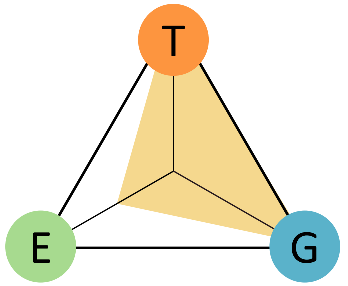
Existing NDC Abstractions

Thread-Level NDC: Offload the entire thread context.



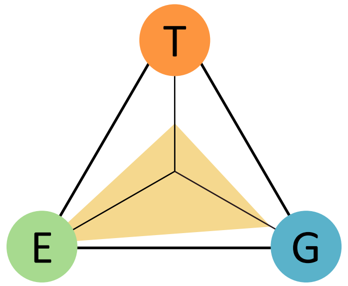
- Examples: TOM [ISCA ’16], AMS [MICRO ’18], …
- ☑ Transparency.
- ☑ Synchronization-Efficiency.
- ☒ Generality: Works best with single data structure.

Inst-Level NDC: Offload short instruction sequences.



- Examples: PIM-Enabled Inst [ISCA ’15], Omni-Compute [ISCA ’19], …
- ☑ Transparency.
- ☒ Synchronization-Efficiency: At least one request per operation.
- ☑ Generality.

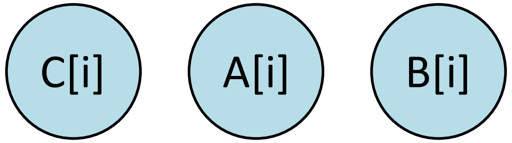
User-Defined Function NDC: Offload user-defined operation.



- Examples: Active Routing [HPCA ’19], Livia [ASPLOS ’20], …
- ☒ Transparency: Requires manually programming.
- ☑ Synchronization-Efficiency.
- ☑ Generality: Depends on the implementation.

Streams: New Abstraction with Near-Data Comp.

```
while (i < N)
  C[i] = A[i] + B[i]
  i++
```



Stream: A decoupled seq. of addr/value.

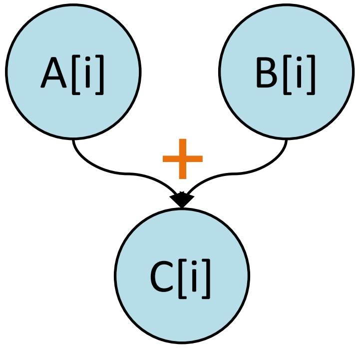
- Prevalent in data parallel workloads.
- Capture long-term patterns for each data structure
  → Rich information for transparent and general offloading.

Near-Stream Computing:

- Computation scheduled along with streams.

Streams: New Abstraction with Near-Data Comp.

```
while (i < N)
  C[i] = A[i] + B[i]
  i++
```



Stream: A decoupled seq. of addr/value.

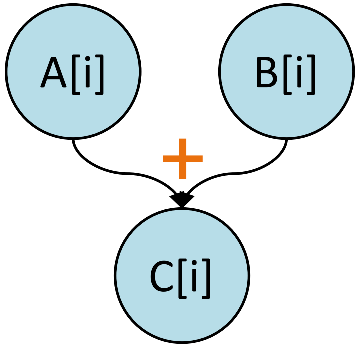
- Prevalent in data parallel workloads.
- Capture long-term patterns for each data structure
  → Rich information for transparent and general offloading.

Near-Stream Computing:

- Computation scheduled along with streams.

Streams: New Abstraction with Near-Data Comp.

```
while (i < N)
  C[i] = A[i] + B[i]
  i++
```



Stream: A decoupled seq. of addr/value.

- Prevalent in data parallel workloads.
- Capture long-term patterns for each data structure
  → Rich information for transparent and general offloading.

Near-Stream Computing:

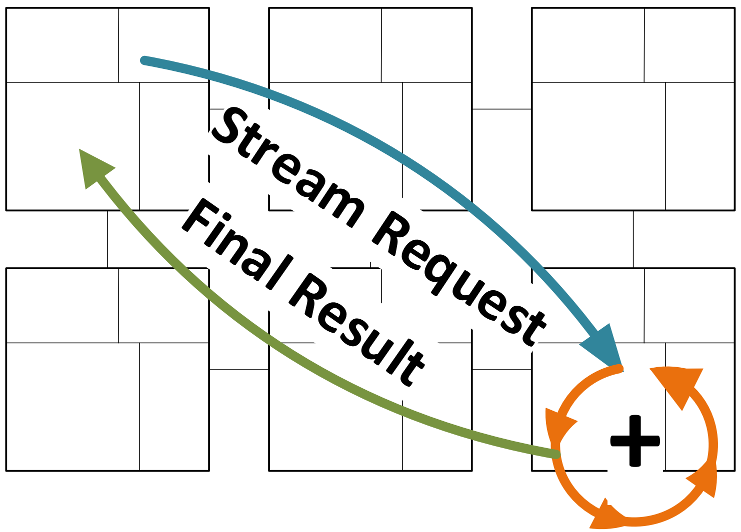
- Computation scheduled along with streams.
- ☑ Transparency: Extensive compiler/ISA support.
- ☑ Efficiency: Coarser-grained range-based sync.
- ☑ Generality: Support combinations of access patterns and ops.

LLVM Compiler + GEM5 Simulator:

- **2.13× speedup over SOTA NDC techniques.**
- **76% NoC traffic reduction.**

Outline

- Problems and Insights
- NDC Taxonomy and Opportunities
- Near-Stream Computing Implementation
- Evaluation



Taxonomy of General Near-Data Computation

Generality.

- Address pattern + compute type.

❑ *Affine*	❑ *Reduce*	❑ *Multi-Op*	❑ *Store*	❑ *Indirect*	❑ *RMW*	❑ *Ptr-Chase*	❑ *Load*
$A[i,j,\ldots]$	$\sigma f({}^*A)$	$A[i], B[i], \ldots$	${}^*A = f()$	$A[\ldots B[i]]$	${}^*A = f({}^*A)$	$A = A.next$	$= f({}^*A)$

```
// Sum Array.
while (i < N)
  S += A[i]
  i++
```

```
// Vector Add.
while (i < N)
  C[i] = A[i] + B[i]
  i++
```

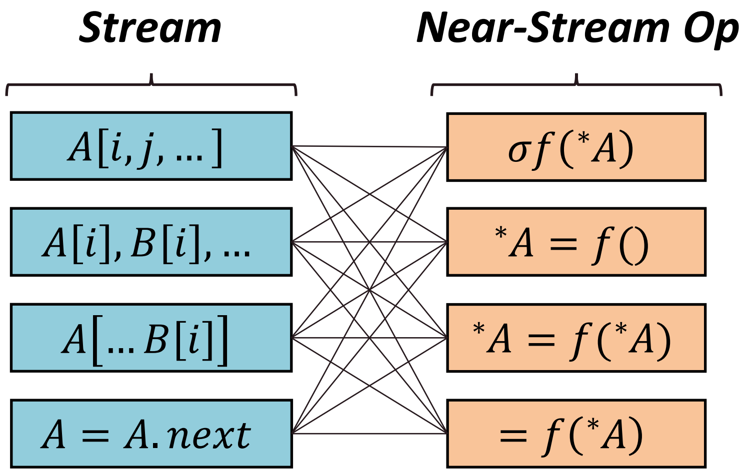
```
// Indirect RMW.
while (i < N)
  B[A[i]]++
  i++
```

```
// Link-List Op.
while (A)
  f(A->value)
  A = A->next
```

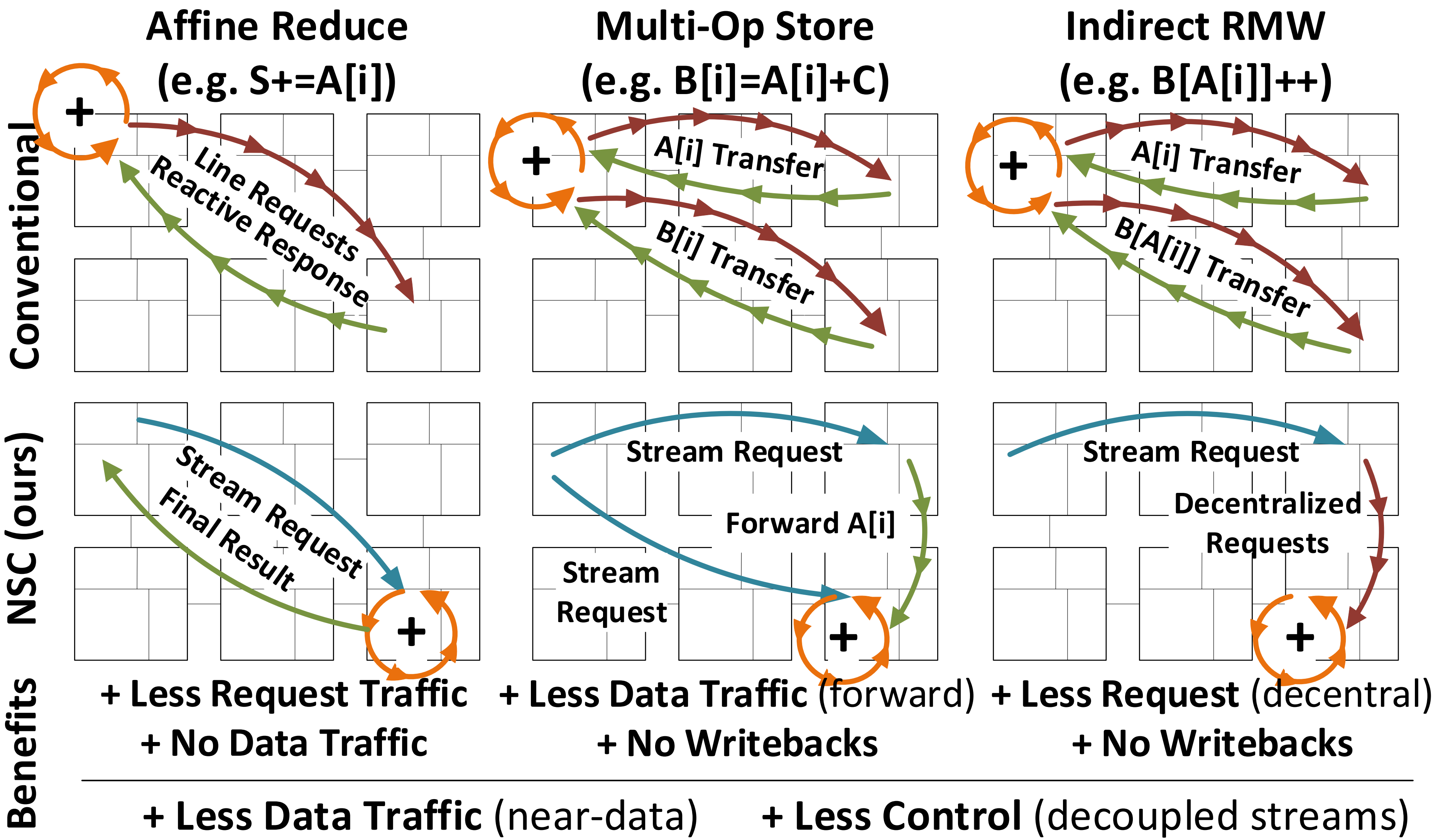
Taxonomy of General Near-Data Computation

Generality.

- Address pattern + compute type.
- ***Support all combinations.***
- Address Pattern → Streams; Compute Type → Near-Stream Operation.

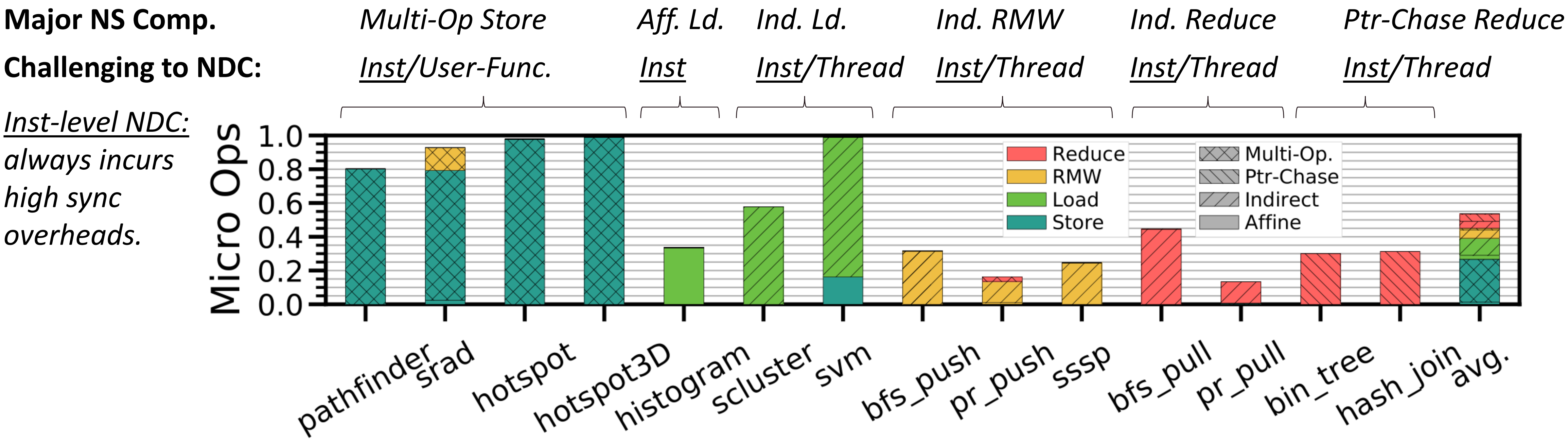


Conventional vs. Near-Stream Computing



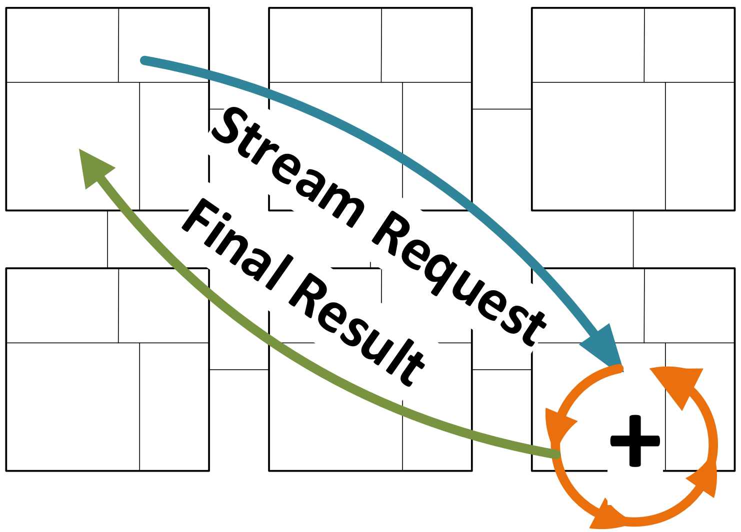
Breakdown of Near-Stream Computation

- 54% of dynamic micro-ops can be classified as near-stream computation.



Outline

- Problems and Insights
- NDC Taxonomy and Opportunities
- Near-Stream Computing Implementation: How to …
  - *Embed near-stream computing in the ISA?*
  - *Preserve sequential semantics?*
  - *Reduce coordination overheads?*
  - *Reuse hardware for computation?*
  - *When to offload computation?*
  - …
- Evaluation



Near-Stream Computing ISA Overview

- Explicitly encode streams and compute dependencies in the ISA.
- Compiler: Recognize streams and associated computation; Transform the program.
- **No manually programming required.**

Original Pseudo Code | *N-S Computing Pseudo Code* | *Stream Dep. Graph*

(a) Vector Sum

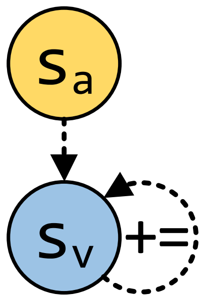
```
while (i < N)
  v += A[i];
  i++;
```

```
s_cfg(s_a=A[i], s_v+=s_a);
while (i < N)
  s_step(i);
v = s_load(s_v);
s_end(s_v, s_a);
```

Config.

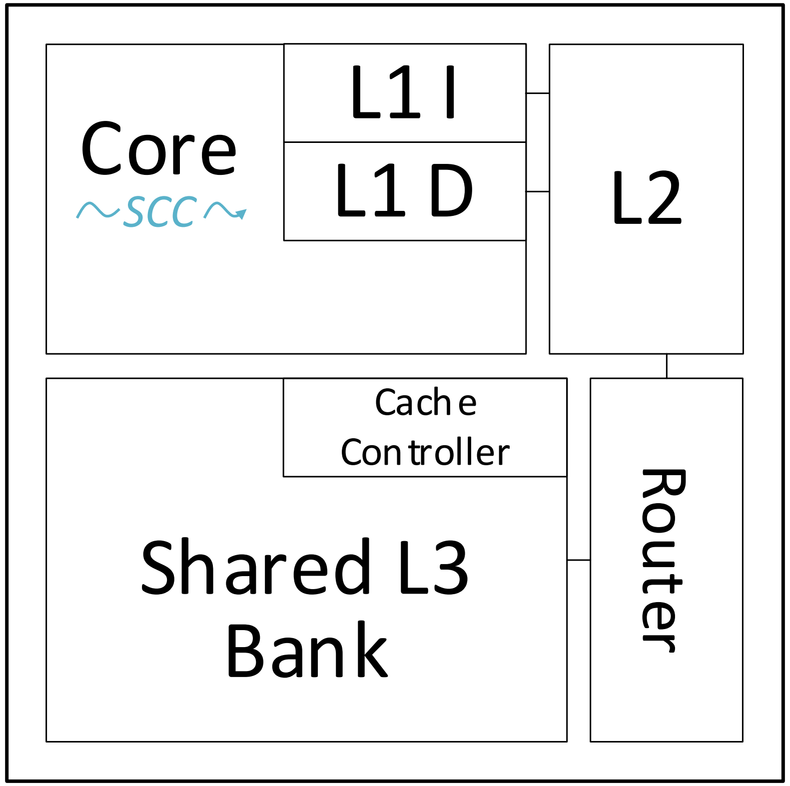
Semantically 1 load and 1 add.

Get final value.



The computation is outlined into a separate function, with the function pointer in the stream's configuration.

How to Compute with Low Overheads?

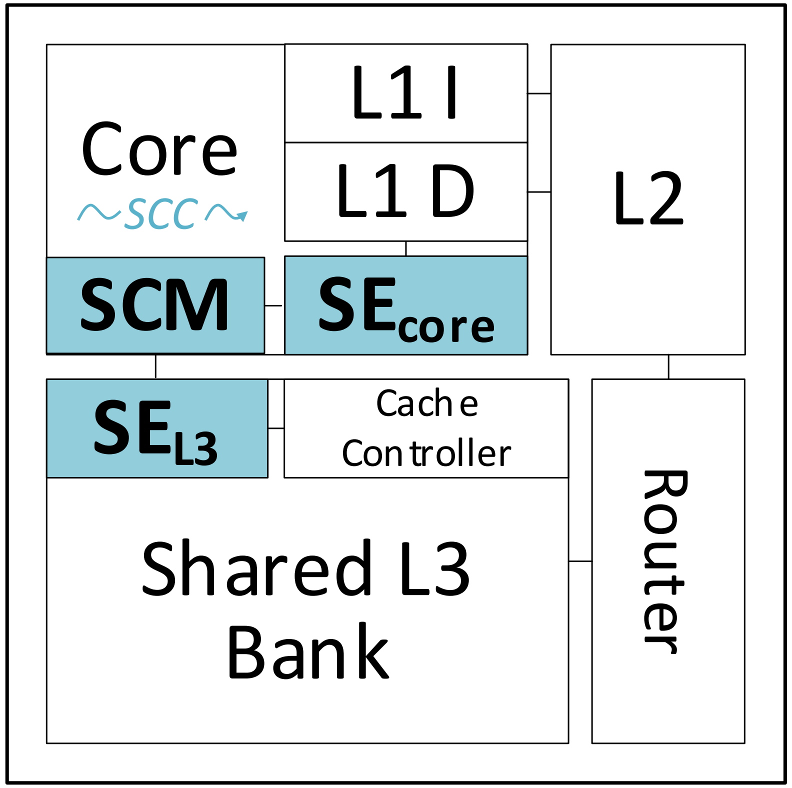


Insight: There is already a core in the remote tile.

Stream Compute Context (**SCC**):

- Light weighted thread context.
- No LSQ entries + Limited ROB entries.
- Looping over configured NSC functions.
- Released when streams are terminated.
- Reuse existing hardware units, e.g., decoder, vector units, …

Architectural Extensions



SE_{core}:

- Mange stream configuration and offload streams.
- Issue flow control credits and check for aliasing.

SE_{L3}:

- Fetch and forward operands to the consuming stream.
- Coordinated by **SE_{core}** for flow control and aliasing check.

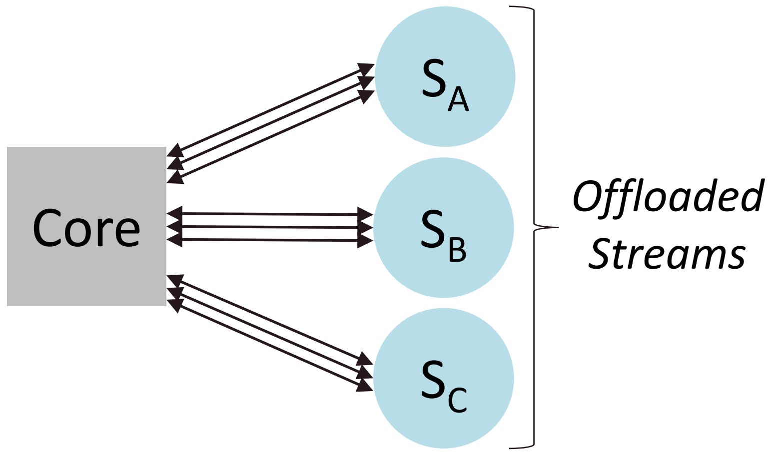
Stream Computing Manager (SCM):

- Configure and terminate the stream computing context (**SCC**) with the near-stream function.
- Simple scalar ops handled by the ALU in **SE_{core}** and **SE_{L3}** Avoid frequently accessing SCC.

Range-Sync: Coarse-Grained Sync. w. Seq. Semantics

Challenges: Efficiently synchronize the core and offloaded streams.

- Flow control scheme to avoid streams running too far ahead/behind.
- Detect possible core-stream and inter-stream aliasing.
- Provide sequential semantics for transparency and context switching.



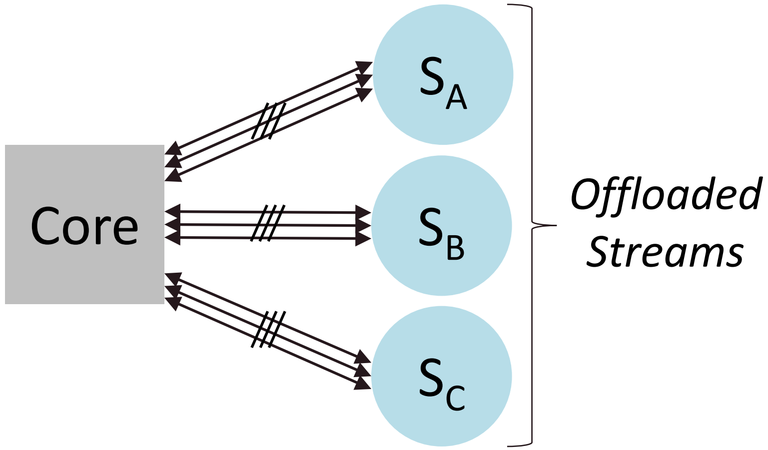
Range-Sync: Coarse-Grained Sync. w. Seq. Semantics

Challenges: Efficiently synchronize between core and streams.

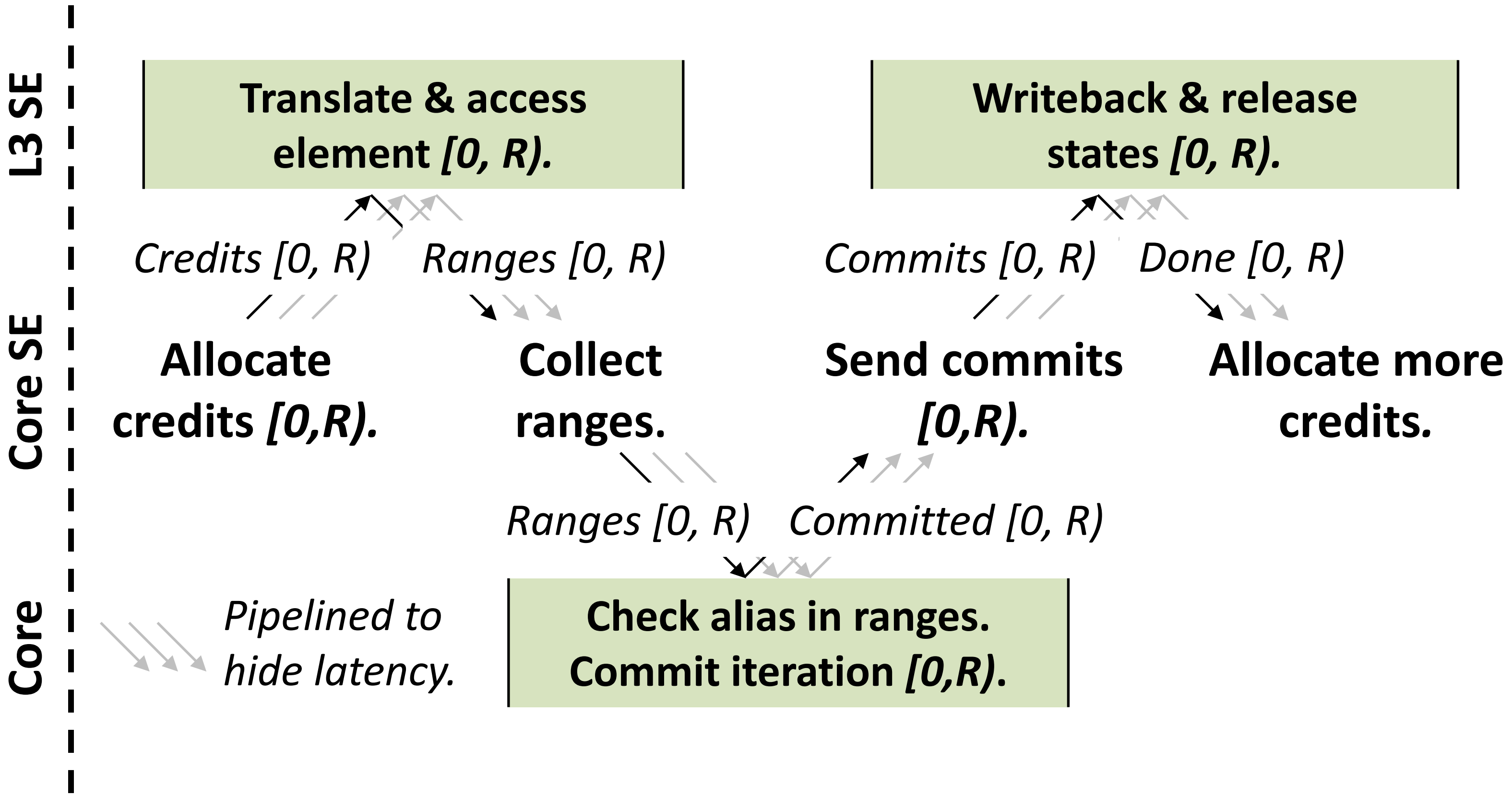
Key Insight: Sync. can be coarse-grained and conservative.

- Streams tend to access individual data structures, with non-overlapping addresses.
- Aliased streams should not be offloaded at all to avoid frequent synchronization.
- Allows false positives to dramatically reduce control overheads.

→ **Range-Sync**: Only sync. every few iterations against range of touched addresses.



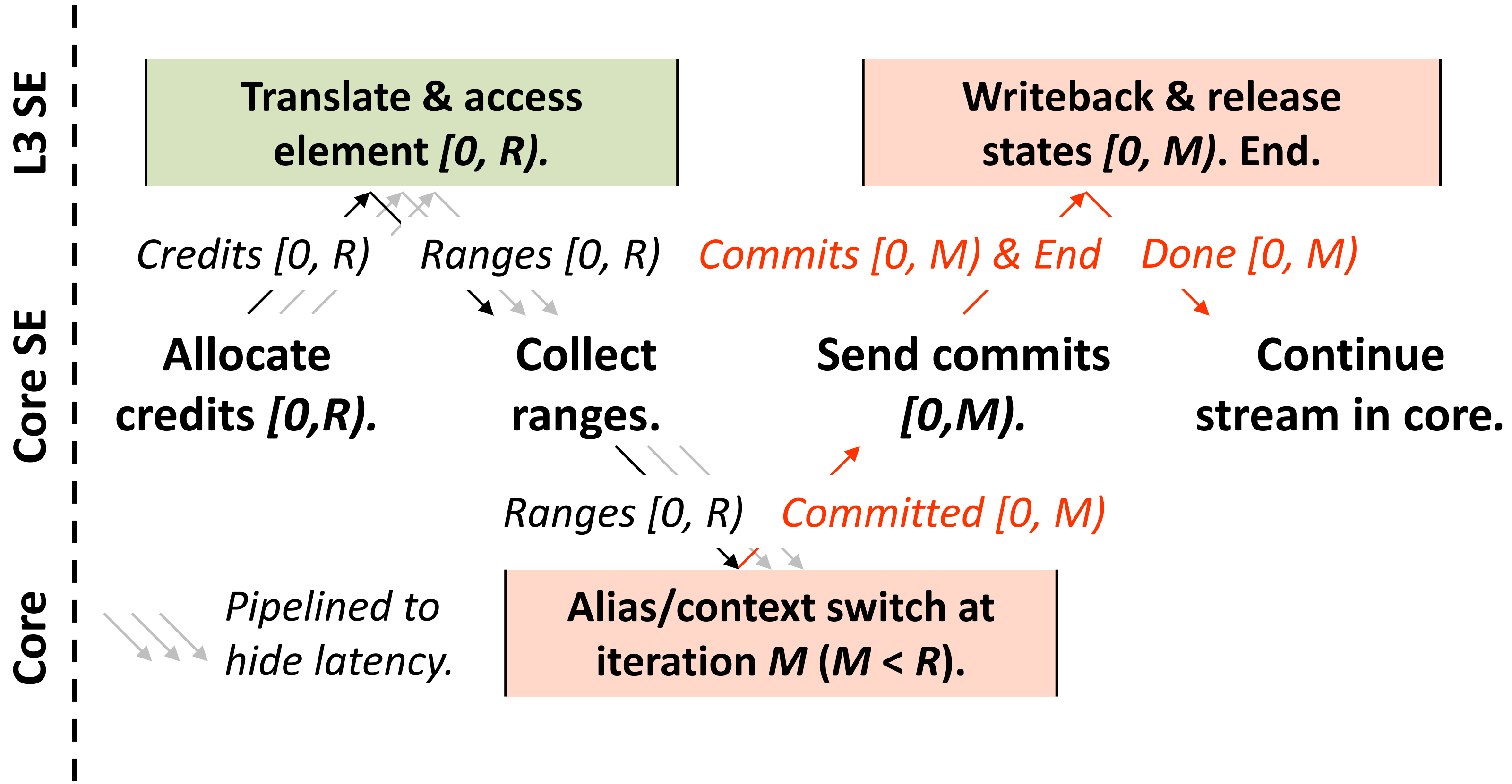
Timeline of Range-Sync: Normal Case



Key Takeaway:

- *Sync every R iterations.*
- *Check alias against ranges.*
- *Overheads: 4/R NoC msg. per affine element.*

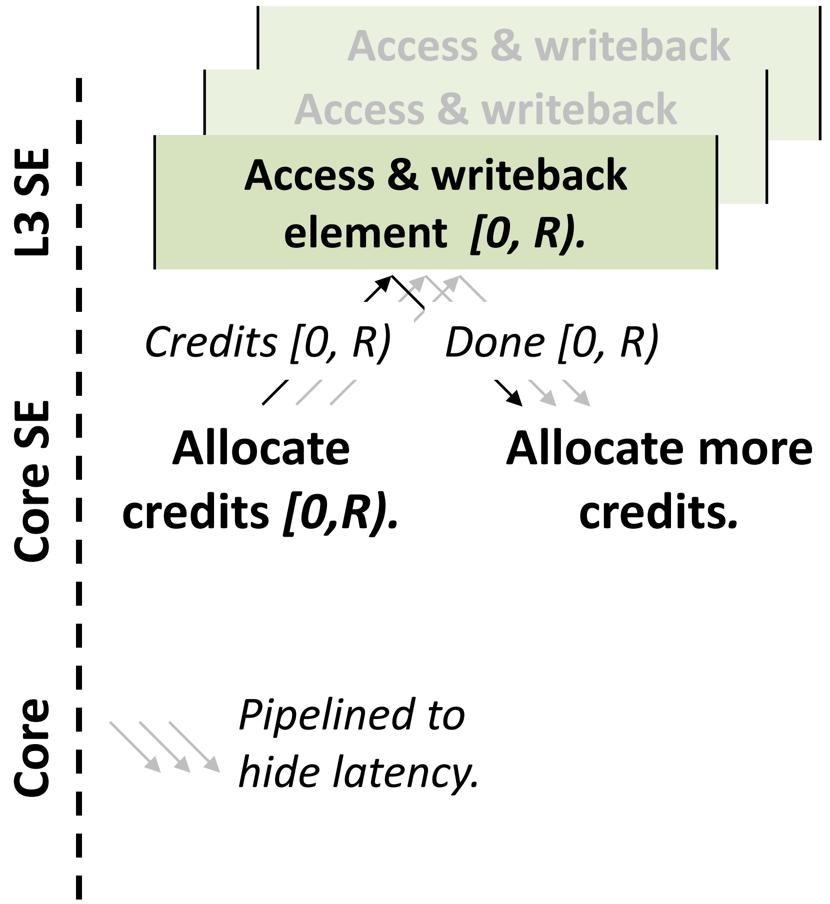
Timeline of Range-Sync: Alias/Context Switch in Core



Key Takeaway:

- *Execution continues at iteration M with streams in core.*
- *Still provide precise states.*

Sync-Free Optimized NSC



Key Takeaway:

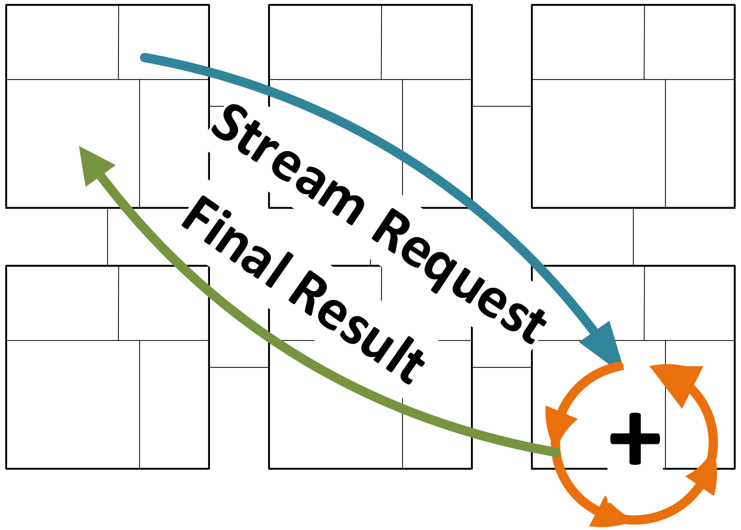
- Pragma disables synchronization.
- Offloaded streams can directly commit.
- Coarse-grained context switch at iteration n*R.
- Fully decoupled loops can be removed.
- → *Simultaneously execute multiple loops.*

More details in the paper:

- Compiler transformation.
- Nested stream configuration.
- Avoid deadlocks.
- Determine when to offload.
- …

Outline

- Problems and Insights
- NDC Taxonomy and Opportunities
- Near-Stream Computing Implementation
- Evaluation



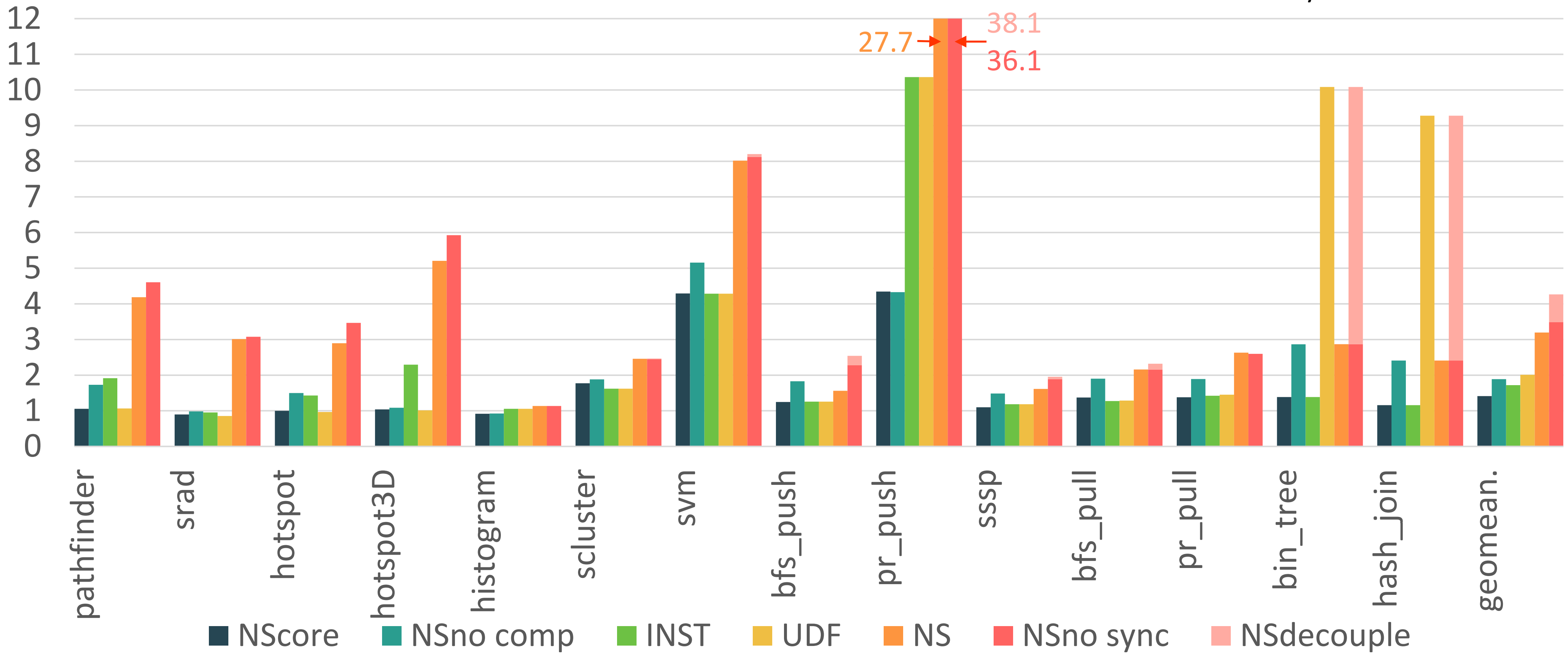
Configurations

- LLVM-based compiler to recognize streams and transform programs.
- Gem5 20.0 cycle-level execution-driven simulator.
- 14 data processing workloads from Rodinia and Gap Graph Suite.
  - Parallelized with OpenMP, with AVX-512 enabled.
- Configurations (see paper for details):
  - 8x8 mesh topology, 3-level MESI, 32kB L1 I/D, 256kB L2, 1MB L3.
  - Base: Baseline cores with L1/L2 prefetchers. No stream support.
  - Inst-Level NDC (INST): Computation is offloaded at instruction level to LLC.
  - User-Def Func. NDC (UDF): Single operand UDF is offloaded at loop level.
  - NS_{core}: Use SE_{core} as a prefetcher at the core [ISCA ’19].
  - $NS_{no\ comp}$: Streams are offloaded to LLC, but no computation [HPCA ’21].
  - **NS: Near-stream computations offloaded to LLC, with range-sync [this work].**
  - **$NS_{no\ sync}$/$NS_{decouple}$: Disable range-sync and fully decouple loops [this work].**

Resembles SOTA NDC:
Omni-Compute [ISCA ’19]
Livia [ASPLOS ’20]

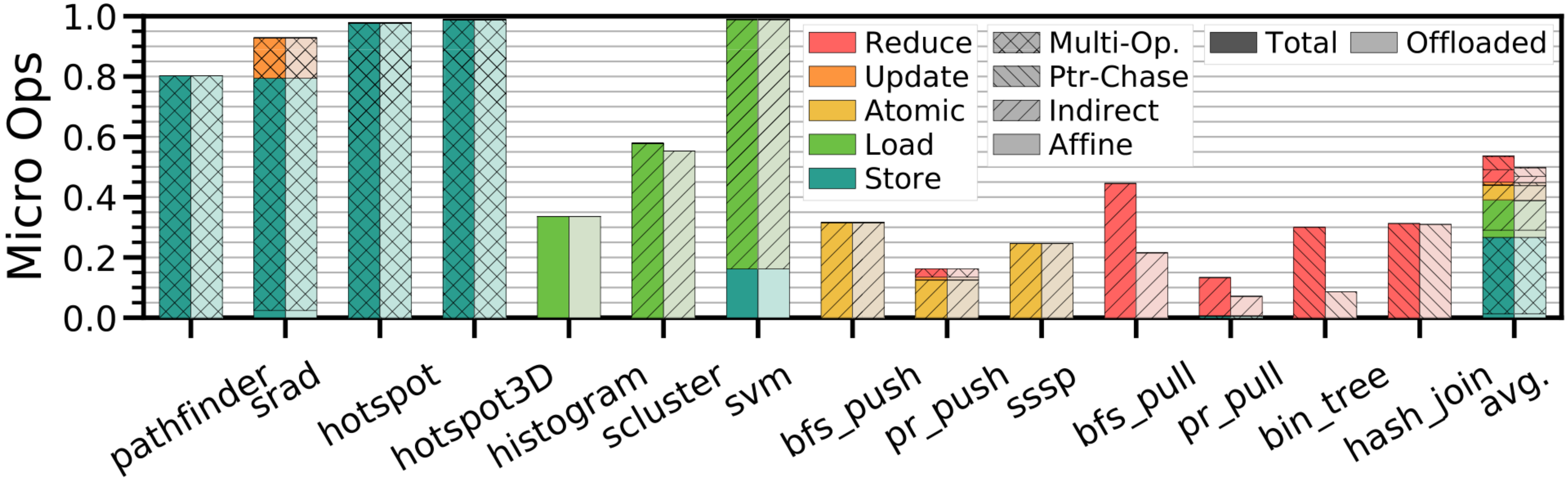
Overall Speedup with OOO8

- NS achieves 1.85x speedup over INST. (2.12x for $NS_{decouple}$ over UDF).
- Range-sync incurs low overheads as NS is only 8% slower than $NS_{no\ sync}$.



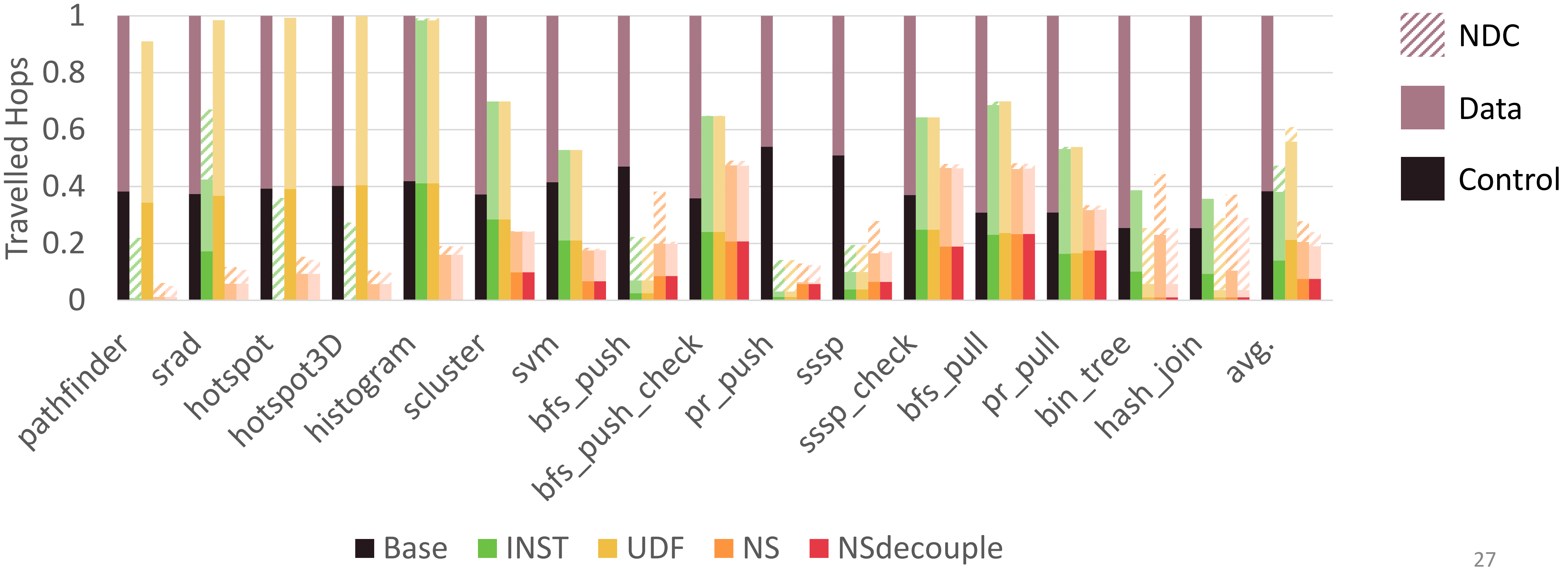
Dynamic Micro Ops Breakdown

- Generality: NSC captures 54% of dynamic micro-ops.
- 93% of possible ops are offloaded.



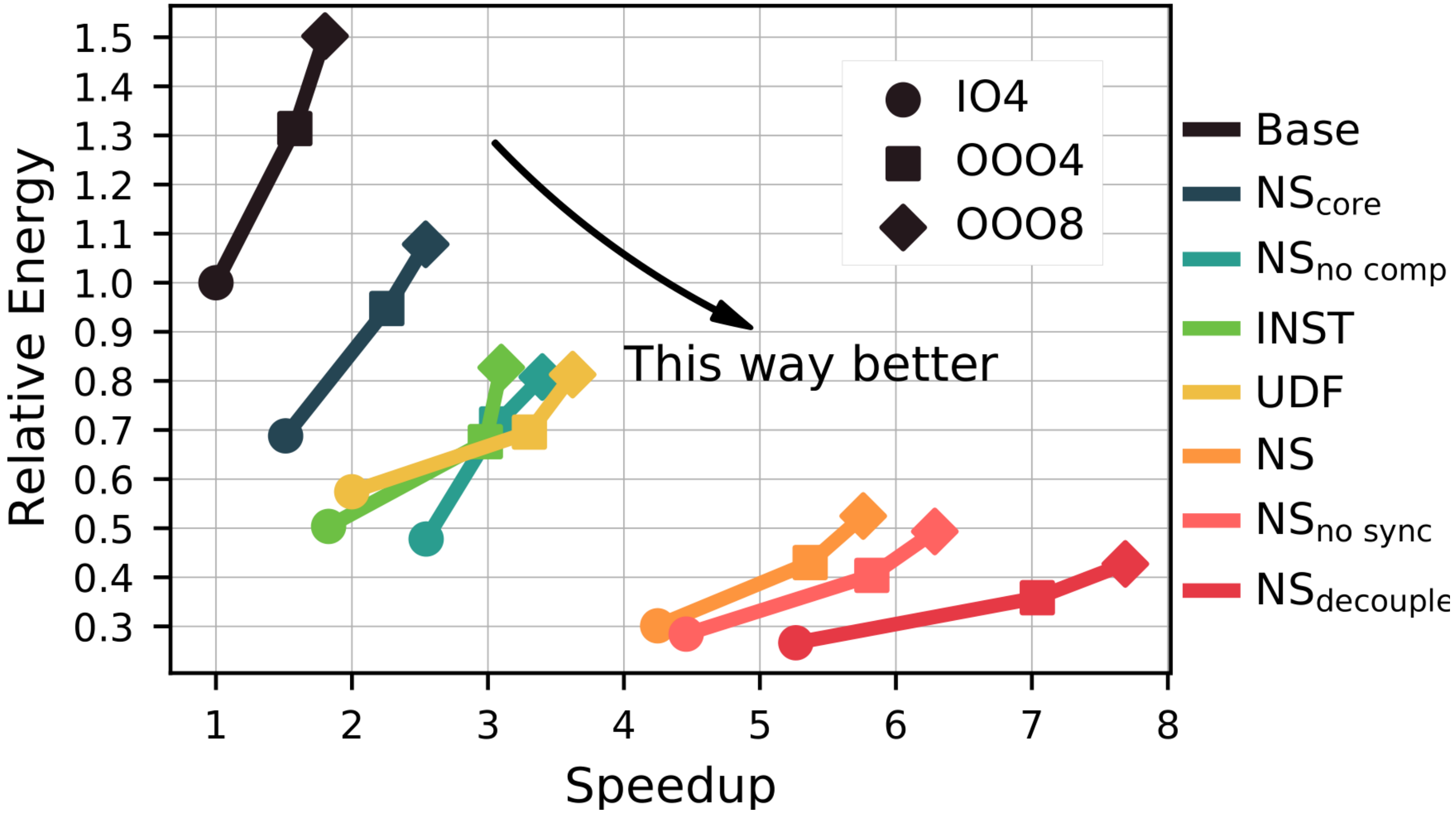
NoC Traffic Breakdown

- INST and UDF reduces NoC traffic by 50% and 40% respectively.
- NS reduces NoC traffic by 69% (76% by $NS_{decouple}$).
- NS offloads at stream granularity and incurs low overheads using range-sync.



Energy vs. Speedup

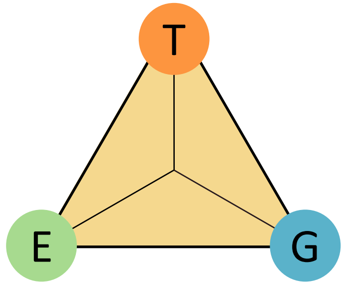
- NS and $NS_{decouple}$ achieves 2.85×/3.52× energy efficiency over OOO8.
- Both small inorder cores and large OOO cores benefit from NSC.



Conclusion

Near-Stream Computing:

- ☑ Transparency: Extensive compiler/ISA support.
- ☑ Efficiency: Coarser-grained range-based sync.
- ☑ Generality: All combinations of address pattern + compute type.



Core Idea: Encode access patterns and compute deps in ISA.

→ Expose rich semantic information to the hardware.

→ Programmer-friendly near-data architectures.