MemGaze: Rapid and Effective Load-Level Memory Trace Analysis

Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Ozgur Kilic, Nathan Tallent, Yasodha Suriyakumar (PNNL & Portland State University)
Chenhao Xie, Andrés Marquez, Stephane Eranian (Google)

Pacific Northwest National Lab

Scalable Tools Workshop 2023

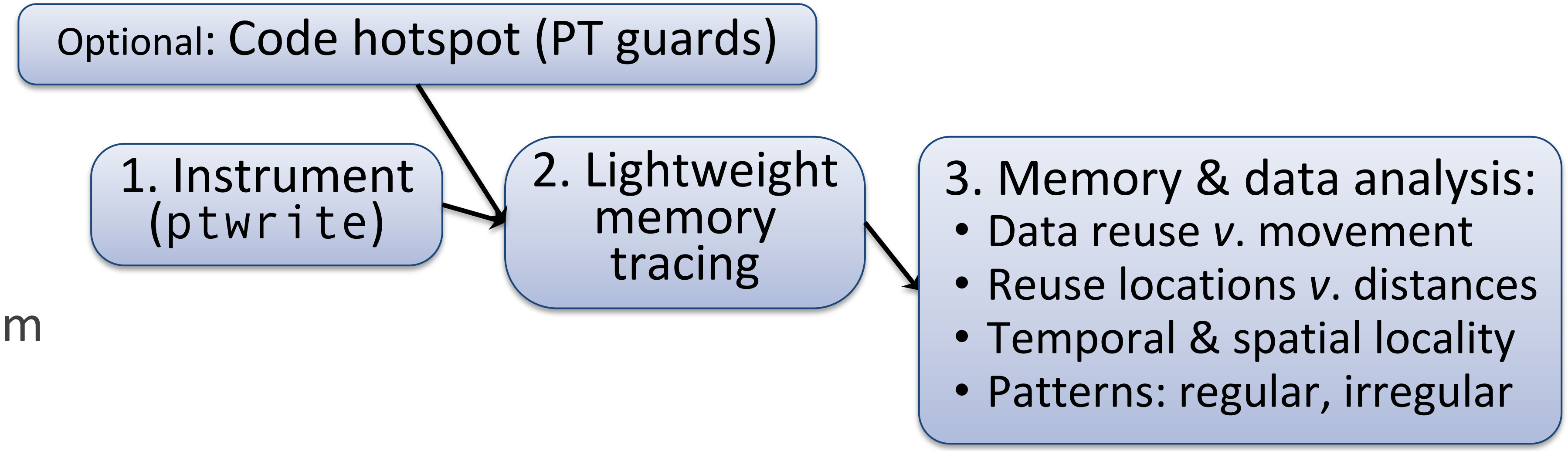
June 20, 2023

MemGaze: low-overhead, high-resolution memory trace analysis

O. Kilic et al. "MemGaze: Rapid and effective load-level memory and data analysis" CLUSTER '22

PNNL

- MemGaze: low-overhead, high-resolution, access sequences
  - Uses Processor Tracing to collect sampled, compressed memory address traces
    - Supported on x86 and ARM
      - Focus: x86 ptwrite
    - Emerging x86 support
      - Server: from Sapphire Rapids
      - Desktop: from Alder Lake; Atom
  - Multi-resolution analysis for
    - accesses vs. memory locations
    - reuse (distance, rate, volume) vs. access patterns
    - spatio-temporal correlations for time vs. location
  - Both trace size and trace resolution are controllable



Space savings: 1% of full trace

Accuracy: Within 25% for sequences; 5% for hotspots

Time overhead, good implementation: 10–35%

Highlights of Processor Tracing

PNNL

- Control flow packets
- Per-core state/buffer
- Write arbitrary packet (64-bits)
- Mask instructions/packets on/off in hardware
  - Enables sampling
- New: Cycle-Accurate Mode
  - Cycle accurate timing
- New: Power events
  - P-states and C-states

Binary instrumentation and Trace compression

- Ensure all instrumentation can be masked by hardware
  - single inline instruction
  - no change of CPU state (e.g., no spilling)

```
ptwrite s
load d ← [s] + o
```

```
ptwrite s1
ptwrite s2
load d ← [s1] + k [s2] + o
```

- Static analysis to classify loads
  - Classification:
    - constant: e.g., stack frame, static data
    - strided: affine
    - irregular: not strided or constant

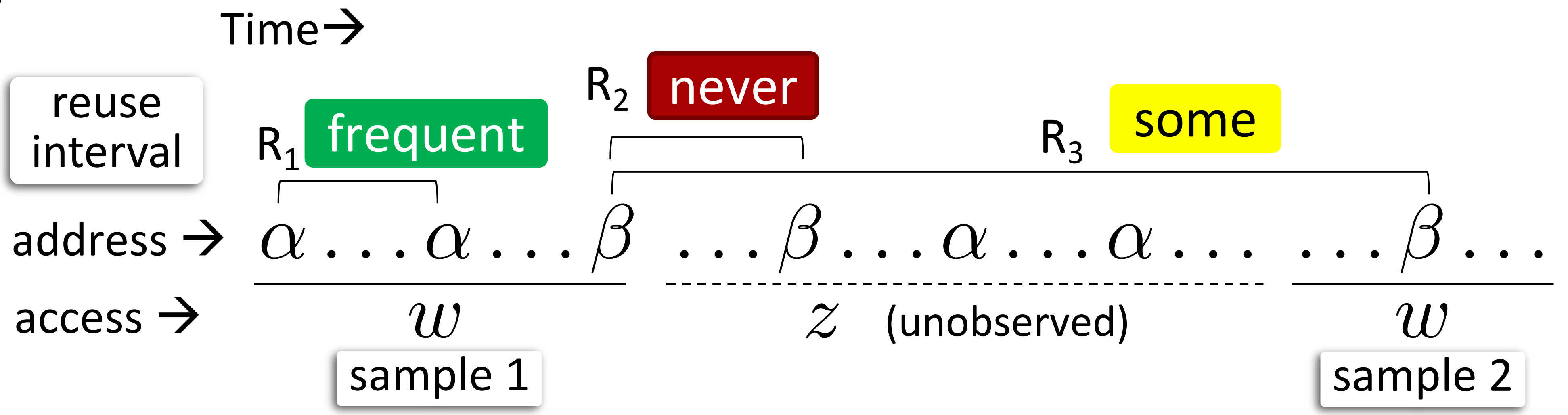
for (i = 0; i < N; i+=2)

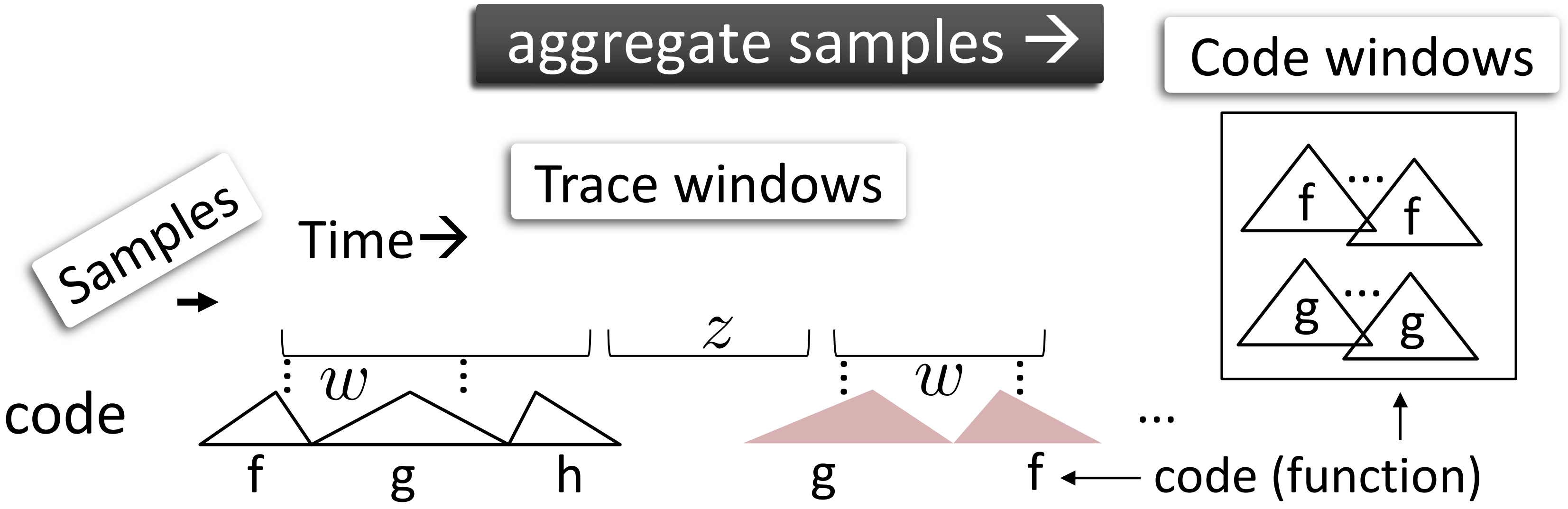
basic block	class	trace annotation	
// a[idx[i]]			
load N	Constant	{ }	no ptwrite; no annotation
load a	Constant	{ }	
load idx[i]	Strided	{strided, 2}	proxy for implied Constant loads
load a[idx[i]]	Irregular	{irregular}	

  - Benefit 1: Compression
    - Indirectly capture Constant loads (often uninteresting) with ptwrite proxy
      - Average of 1.2x (O3) and 2x (O0) space savings
    - For correctness, ensure basic block has at one ptwrite proxy
  - Benefit 2: Rapid trace analysis
    - Load classes → automatic access patterns, reduces time and overhead of subsequent analysis

Sampled memory traces with Processor Tracing

- Processor Tracing cannot collect exhaustive traces
  - Unpredictable data drops when buffers fill (e.g., kernel to user)
  - Unmanageably large, O(GB/s)
- Sampled trace: sequence of *w* seen & *z* unseen accesses
  - Control buffer size and period between samples
- Question: Blind spots?
  - R1 *frequently* observed
  - R2 *never* observed
  - R3 *sometimes* observed
- Reduce error with sample aggregation
  - Code windows aggregate samples
  - Source code attribution of instrumented code

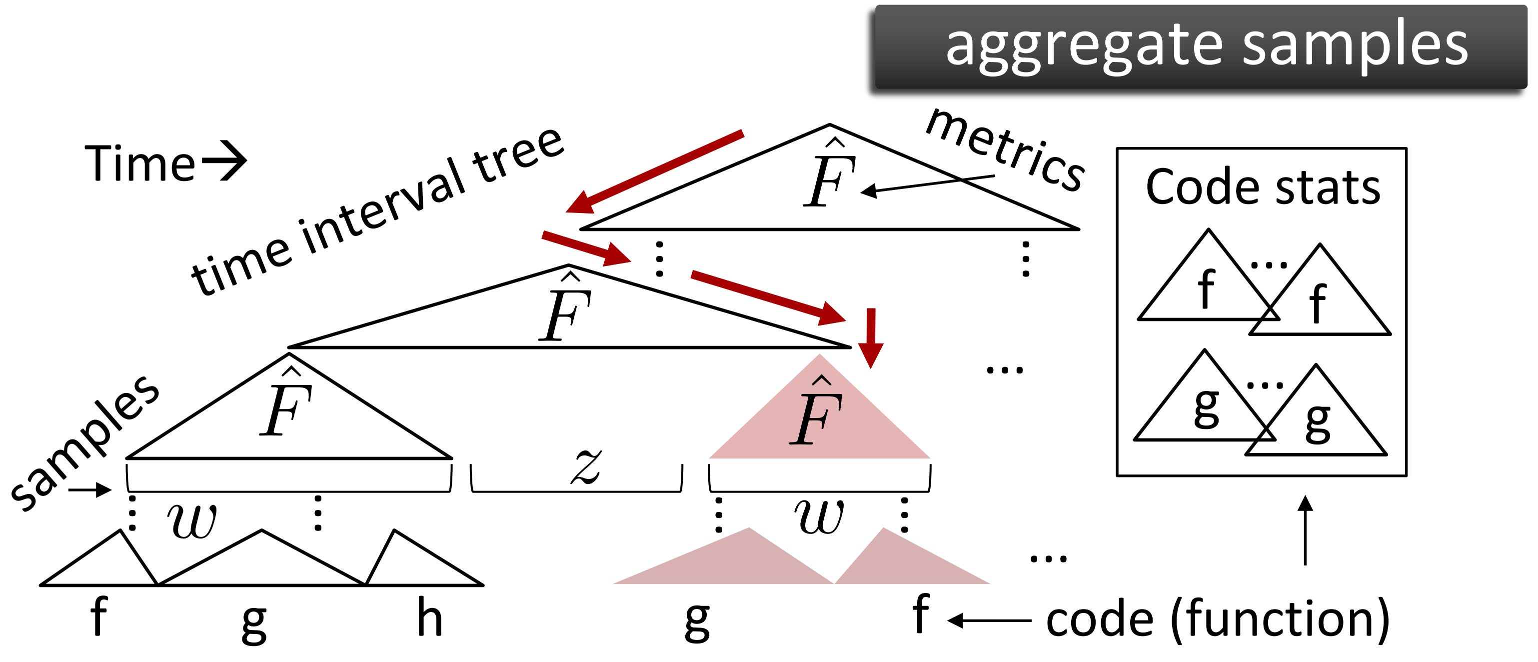




Analyzing memory *operations* over time

PNNL

- Top-down analysis with tree structure
  - root: entire execution
  - interior: decreasing time intervals
  - leaves: samples
- Guidance with data locality metrics
- Code structure within sample
  - support line mapping for PT instrumentation



DarkNet (CNN) Inference (gemm): Data locality over time, hot access intervals

Access Interval	AlexNet F	AlexNet footprint rate	AlexNet reuse distance	AlexNet $\mathcal{A}$	ResNet152 F	ResNet152 footprint rate	ResNet152 reuse distance	ResNet152 $\mathcal{A}$
0	28M	**0.475**	**0.01**	30K	639M	**0.747**	**0.47**	286K
1	55M	**0.675**	**0.02**	30K	772M	**0.799**	**0.57**	293K
2	89M	**0.983**	**0.02**	25K	640M	**0.617**	**2.71**	302K
3	64M	**0.794**	**0.14**	26K	620M	**0.599**	**2.62**	304K
4	39M	**0.489**	**1.64**	29K	591M	**0.574**	**2.69**	302K
5	55M	**0.627**	**1.66**	26K	638M	**0.618**	**2.65**	302K
6	41M	**0.493**	**1.66**	29K	648M	**0.625**	**2.63**	304K
7	38M	**0.644**	**1.49**	17K	549M	**0.514**	**2.66**	312K

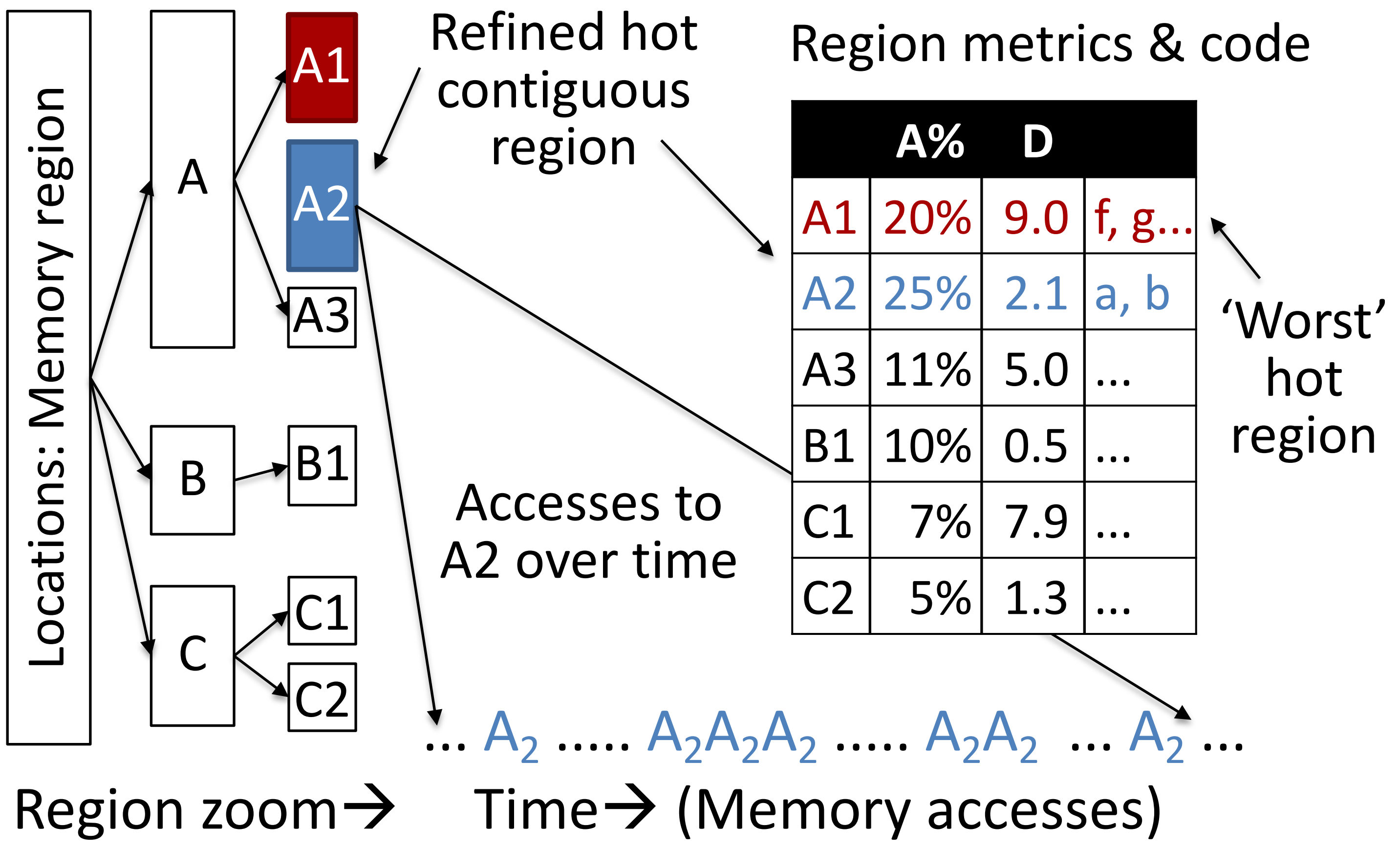
Time ↓

- AlexNet: Δ*F* changes with layer (conv., fully, pooling) vs. ResNet's consistency
- ResNet: ΔF ≈decreases: matrix dims change (N, decreases; K, small increase)

- *D* ≈increases time: matrix dim N decreases with higher level CNN filters

Analyzing memory *locations* over time

- Top-down analysis with tree structure
  - root: all memory locations
  - leaves: refined, hot contiguous regions
- Guidance with data locality metrics
  - Spatio-temporal analysis
- Associate region with code and object



Darknet: Spatio-temporal reuse of hot memory (64 B)

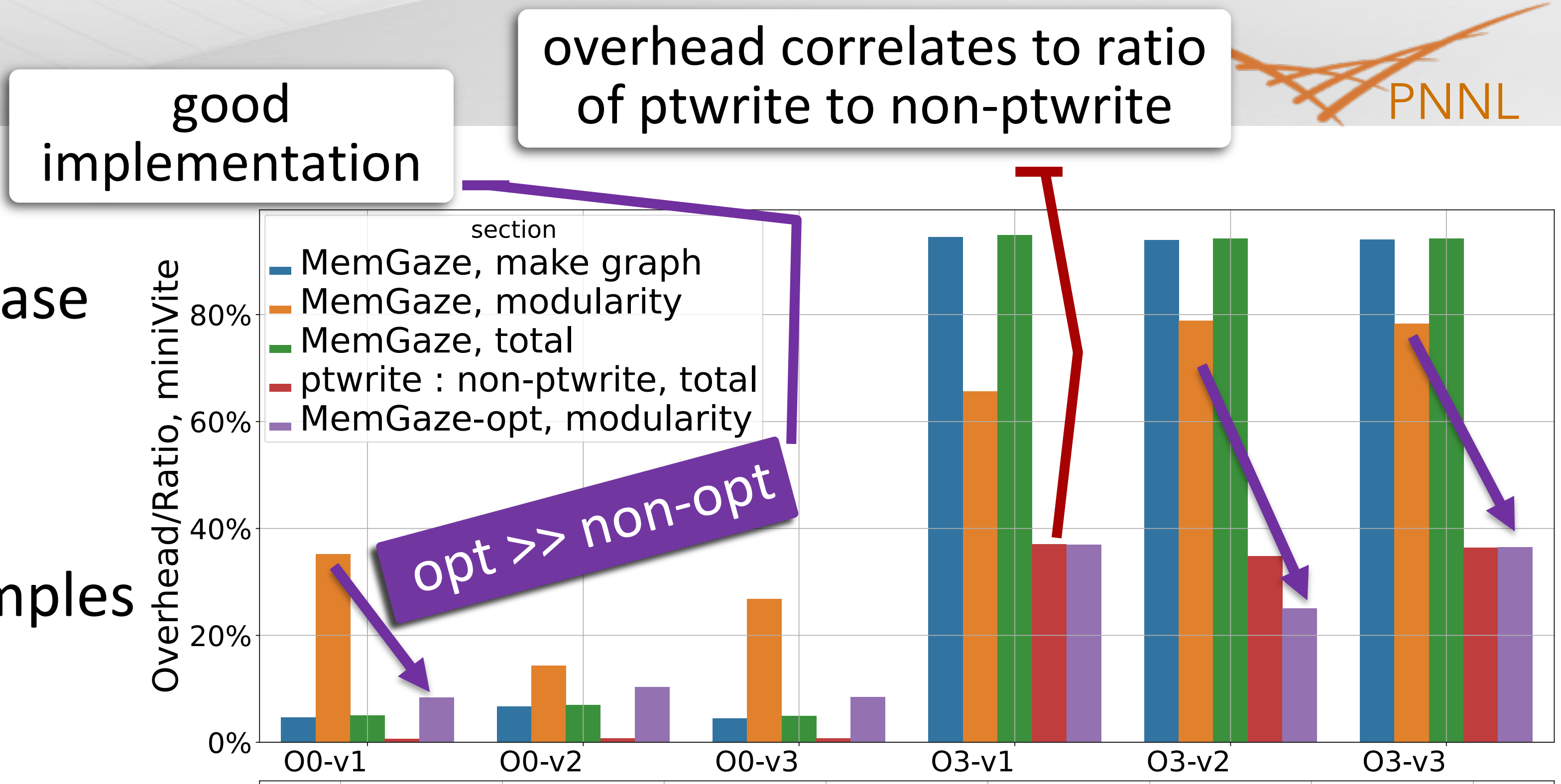
		Reuse distance	Size	Acceses	
Object	Model	Reuse (D)	# blocks	A	A / block
gemm's **A**,**B**,**C**	AlexNet	0.76	66048	977K	14.8
gemm's **B**	ResNet152	0.01	38400	598K	15.6
hot region in im2_col	AlexNet	1.87	8192	167K	20.4
	ResNet152	2.54	3328	7K	1.9

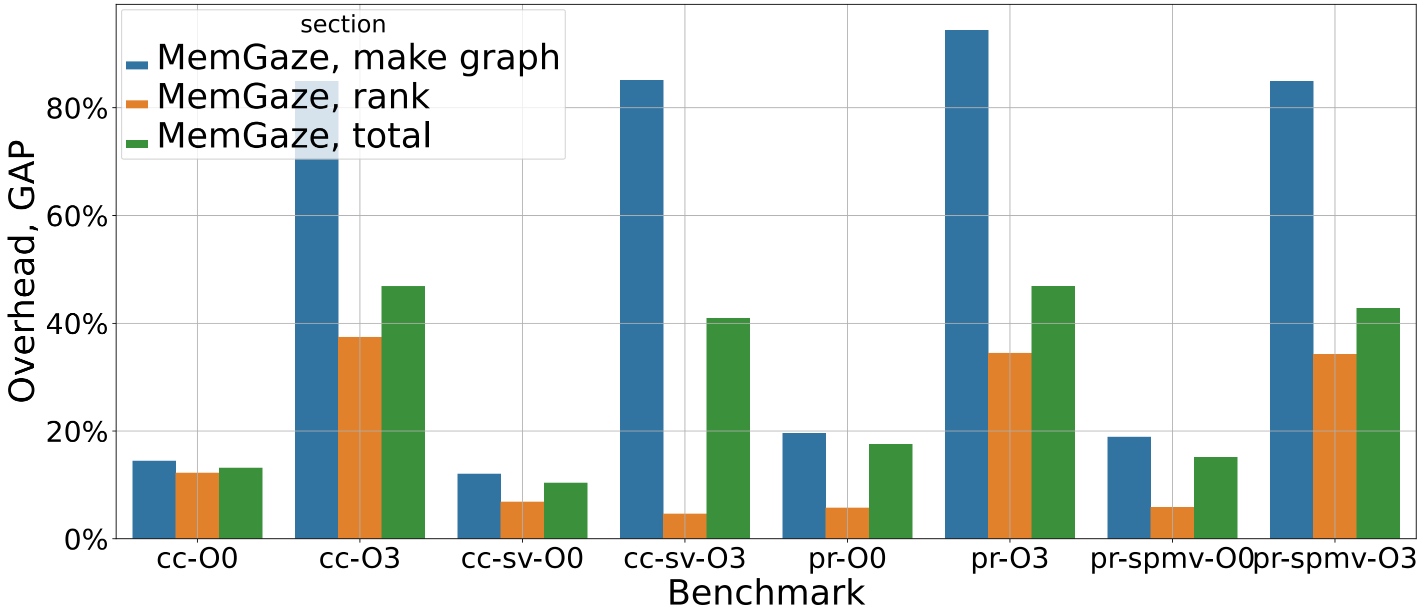
gemm matrices → hottest data

Reuse distance depends on data & neural network

Time overhead for tracing

- Overhead proportional to ptwrites
  - depends on code generation, application/phase
- MemGaze: 5-7×, 10-95%
  - suboptimal implementation (current)
  - PT runs *continuously*; retains data during samples
- MemGaze-opt: 7× → <10%, 80% → 35%
  - PT enabled only during sample
  - user space implementation (proof-of-concept)
- Times for instrumentation & post-mortem analysis in paper
  - suboptimal implementations
  - reasonable times b/c of reduced trace size





For a good implementation (PT only during samples), overhead is 10-35% on memory intensive regions

Early eval (Atom)

DarkNet	MemGaze	MemGaze-opt
AlexNet/ResNet	5× / 7×	10% / 2%

Space reduction for traces

PNNL

- Full trace – not exhaustive due to drops!
  - Rec: recorded, with throttling and drops
  - All: adjusted with drop information
  - All$^+$: full size (includes 'Constant' loads)
- MemGaze trace: ≈1% of full (All$^+$ vs All)
  - sampled and compressed
- Trace compression saves...
  - High compiler opt (O3): 1.2×
  - No compiler opt (O0): 2×

MemGaze trace is ≈1% of full

Benchmark	Full (GB)			MemGaze	Ratio (%)		
	Rec	All	All$^+$	(MB)	Rec	All	All$^+$
all μbench-O0 (1×)	1.9	1.9	3.5	63	3.3	3.3	1.8
all μbench-O3 (1×)	1.9	1.9	1.91	20	1.1	1.1	1
all μbench-O3	112	112	113	865	0.8	0.8	0.7
miniVite-O0-v1	77	163	316.5	1620	2.1	0.9	0.5
miniVite-O0-v2	71	198	387.9	1697	2.4	0.9	0.4
miniVite-O0-v3	79	150	292.7	1660	2.1	1.1	0.6
miniVite-O3-v1	19	41	41.1	310	1.6	0.8	0.7
miniVite-O3-v2	22	43	54.9	310	1.4	0.7	0.6
miniVite-O3-v3	13	23	29.4	341	2.6	1.5	1.1
GAP-cc-O0	2.3	3.4	6.6	355	15.4	10.4	5.3
GAP-cc-O3	4.9	7.9	9.5	31	0.6	0.4	0.3
GAP-cc-sv-O0	4.4	6.4	12.5	377	8.6	5.9	3
GAP-cc-sv-O3	6.7	10.8	13	35	0.5	0.3	0.3
GAP-pr-O0	5.1	7.5	14.6	377	7.4	5.0	2.5
GAP-pr-O3	5.4	7.9	9.5	35	0.7	0.4	0.4
GAP-pr-spmv-O0	6.3	8.9	17.4	385	6.1	4.3	2.2
GAP-pr-spmv-O3	6.5	10.1	12.1	36	0.6	0.4	0.3
Darknet-AlexNet	4.6	11.2	16.9	71	1.6	0.6	0.4
Darknet-ResNet	29	59	66	748	2.6	1.3	1.2

Validation of data locality metrics

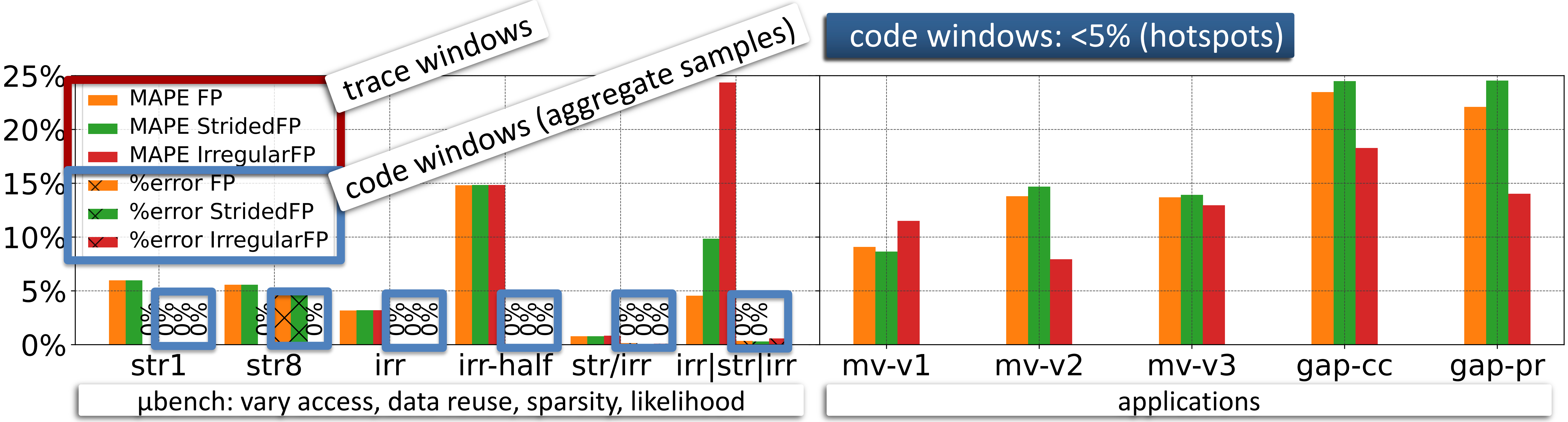
PNNL

- Compare against full traces for microbenchmarks and applications
  - µbench: no drops by inserting OS sleep after each load
  - apps: 10× more samples (full not feasible due to space)
- Trace and Code windows
  - Trace: mean absolute % error (MAPE) for histograms of trace windows: 1-25%
  - Code: % error: <5%

For sampled, compressed trace (≈1% of full)...

trace windows: 1-25% MAPE (varying dynamic sequence lengths)

code windows: <5% (hotspots)



Case studies (Details in paper)

PNNL

- Graph clustering (Louvain Community Detection, miniVite): vary data structures
  - Vary hash table implementations, “open” (array + lists) vs. “closed” (array)
  - Vary compiler optimization levels
- Deep N. Net Inference (DarkNet): vary models
  - AlexNet vs. ResNet
- Graph analytics (GAP): vary algorithms
  - PageRank: Gauss-Seidel vs. Jacobi-style
  - Connected Component: Afforest vs. Shiloach-Vishkin
- Observations:
  - Need time-based & location-based analysis
  - Need complementary metrics and views

All use OpenMP threading

Several analyses w.r.t. time and location

access...
hotness
distance
volume
pattern
rate

Symbol	Analysis
A	Accesses (memory)
D	Spatio-temporal block reuse distance
F	Footprint
F_{str}, F_{irr}	Footprint with strided/irregular access
$F_{str\%}$, $F_{irr\%}$	Fraction of strided/irregular footprint
$A_{const\%}$	Fraction of accesses to ‘constant’ data
ΔF	Footprint growth rate; footprint per access
$\Delta F_{str\%}$, $\Delta F_{irr\%}$	Fraction of strided/irregular footprint growth

Graph Clustering (miniVite): Vary hash table implementations

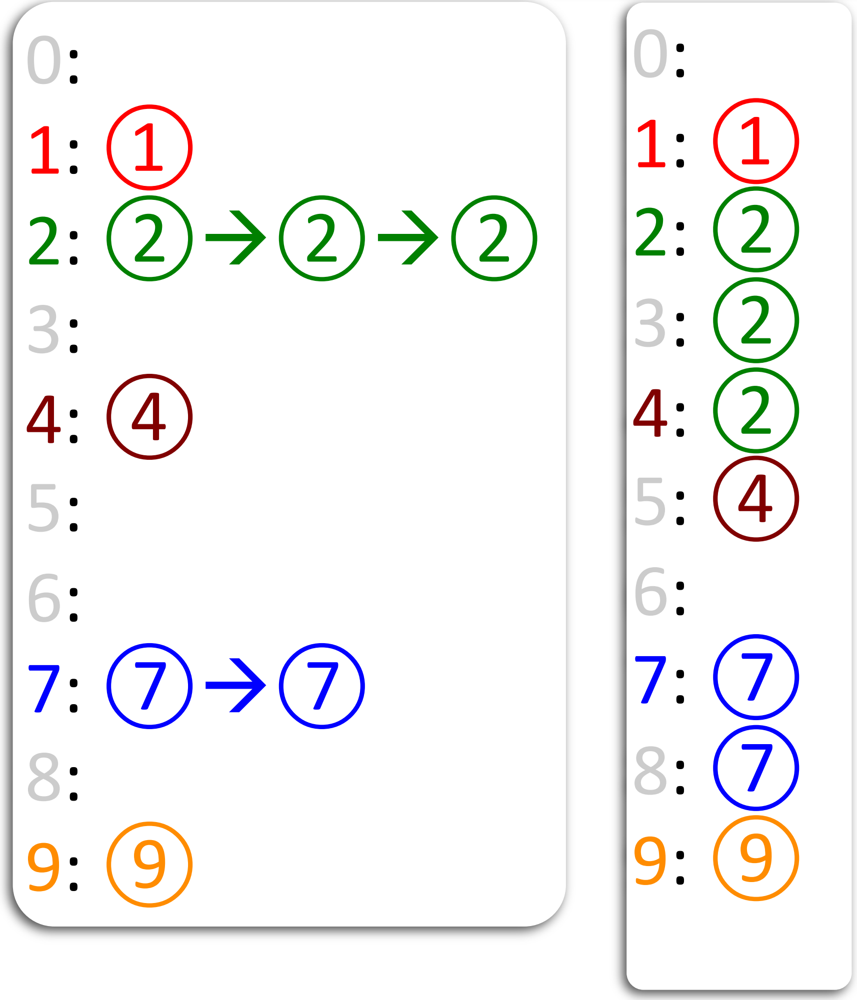
PNNL

V1: C++ map (unordered)

V2: hopscotch default size

V3: hopscotch 'right' size (vertex degree)

Open (C++ map) | Closed (Hopscotch)



Run times	
v1	8.60 s
v2	5.15 s
v3	3.88 s

Data locality of hot function accesses

Function	Variant	F	ΔF	$F_{str\%}$	$\mathcal{A}$
buildMap (make *map*)	v1	2.3G	0.156	66.4	291K
	v2	2.1G	0.151	66.9	273K
	v3	2.1G	0.160	66.8	270K
map.insert	v1	>0.7G	**0.011**	**73.3**	**106K**
	v2	2.4G	**0.003**	**93.7**	**318K**
	v3	0.5G	**0.009**	**92.8**	**67.8K**
getMax (use *map*)	v1	0.4G	**0.150**	**0.5**	**44.7K**
	v2	1.3G	**0.040**	**98.4**	**182K**
	v3	1.5G	**0.040**	**97.8**	**194K**

better pattern

hopscotch must manage size!

Spatio-temporal reuse, hot memory (64 B block)

Object	Variant	**Reuse** (D)	# blocks	A	A / **block**
map (hash table)	v1	**2.65**	768	55K	**71.9**
	v2	**2.79**	768	119K	**155.2**
	v3	**1.97**	768	85K	**111.3**
remote edges of local vertices	v1	**8.71**	4864	24K	**4.9**
	v2	**4.90**	4864	19K	**3.9**
	v3	**3.32**	4864	19K	**3.9**
other objs in buildMap (from caller)	v1	0.37	104K	19235	0.2
	v2	0.15	101K	21362	0.2
	v3	0.24	110K	22306	0.2

Location analysis clearer than time

Sparse structures → smaller footprint, more irregular
Dense structures → larger footprint, more regular, but...

GAP PageRank and Connect Components (CC): Vary algorithms

PNNL

Spatio-temporal reuse of hot memory (64 B block)

Object	Algorithm	Reuse (Avg, Max)		Accesses A	A/block	Time
o-score	pr	**1.13**	**152**	**64K**	0.76	57.2 s
o-score	pr-spmv	**2.41**	**132**	**82K**	1.14	80.1 s
cc	cc	**5.21**	**154**	**581K**	8.87	2.7 s
cc	cc-sv	**0.83**	**36**	**476K**	8.65	45.5 s

For PageRank, spatio-temporal shows difference

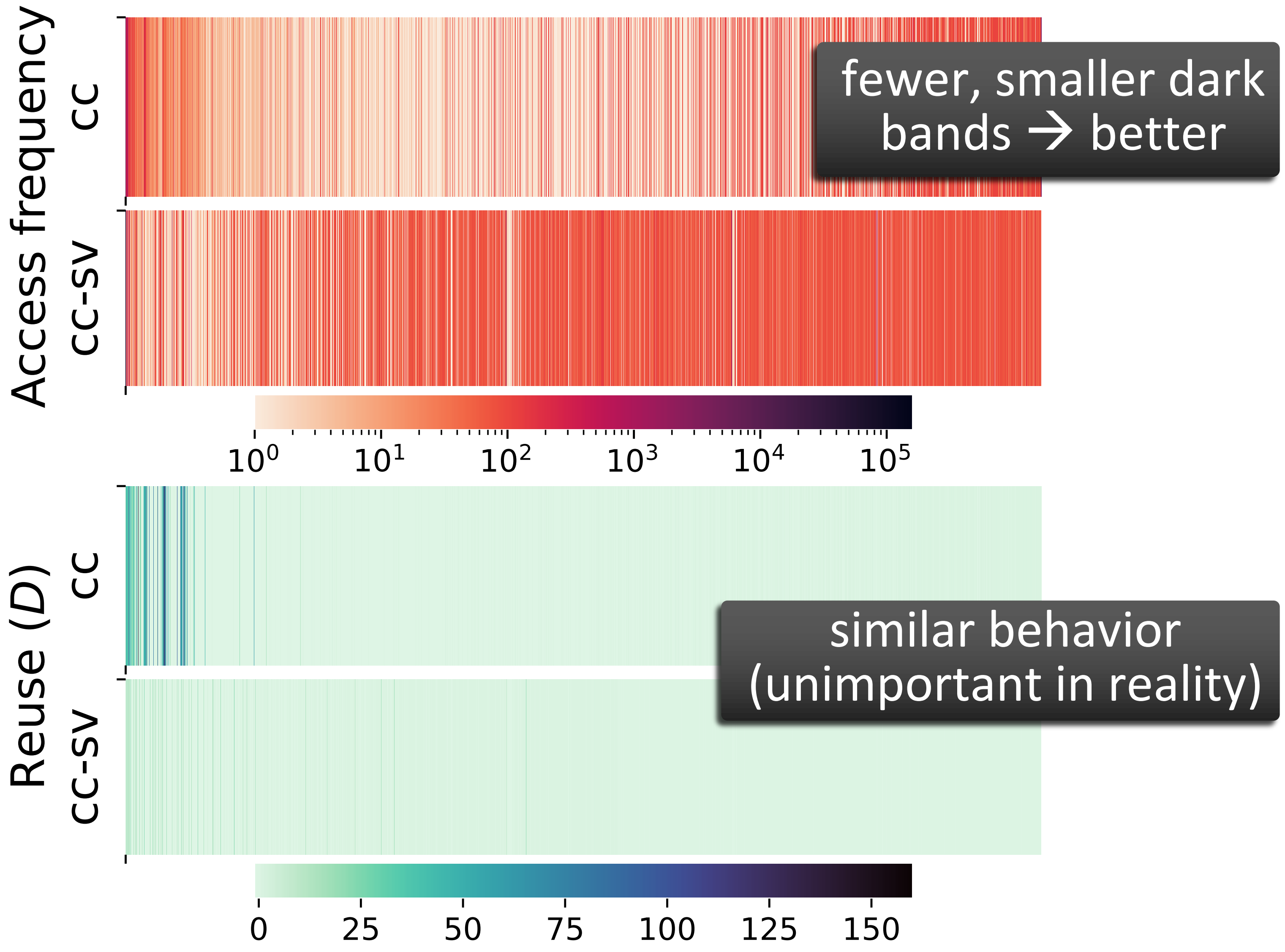
- pr (optimized) vs. pr-spmv
  pr updates o-score ‘now’ vs. next iteration
- Reuse and Δ*F* (not shown) are better
- Accesses are better → fewer iterations

For CC, averages are misleading → heatmaps

- cc (optimized) vs. cc-sv
  cc → more accesses; can improve locality
- Metrics (D, Δ*F*, $F_{irr\%}$) for cc are worse...

Heatmaps!

Distribution of spatio-temporal metrics



Need many angles & many resolutions!

Conclusions

PNNL

- Processor Tracing effective for low-overhead, high-resolution memory analysis
  - accesses (operations) vs. memory locations
  - accesses vs. spatio-temporal reuse
  - reuse (distance, rate, volume) vs. access patterns
- Sampled traces are 1% of full ones → MB vs. GB-TB
- With a straightforward optimization, time overhead is 10-35% vs. 100× or more
  - PT generalizes much performance and state telemetry (without interrupts)
- Analyses explain effects of...
  - different data structures, algorithms, and data sets
  - different access patterns (strided, irregular), that both have ‘good’ spatio-temporal locality
- Future work: hardware/software co-design, automated diagnostics, ...

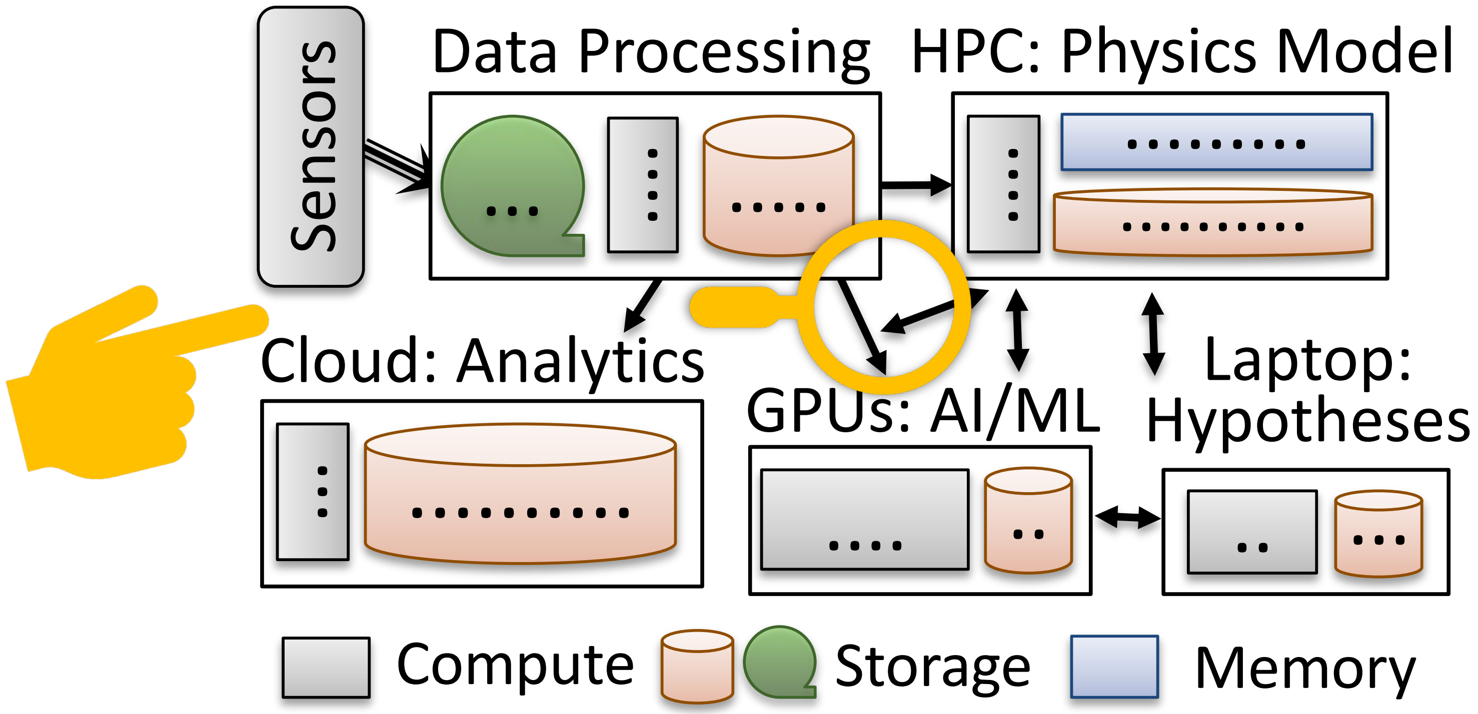
`github.com/pnnl/memgaze`

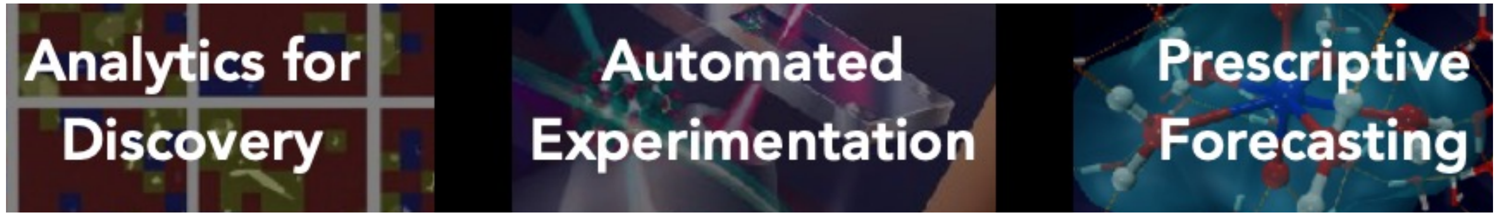
Opportunities: Job, intern, and collaboration

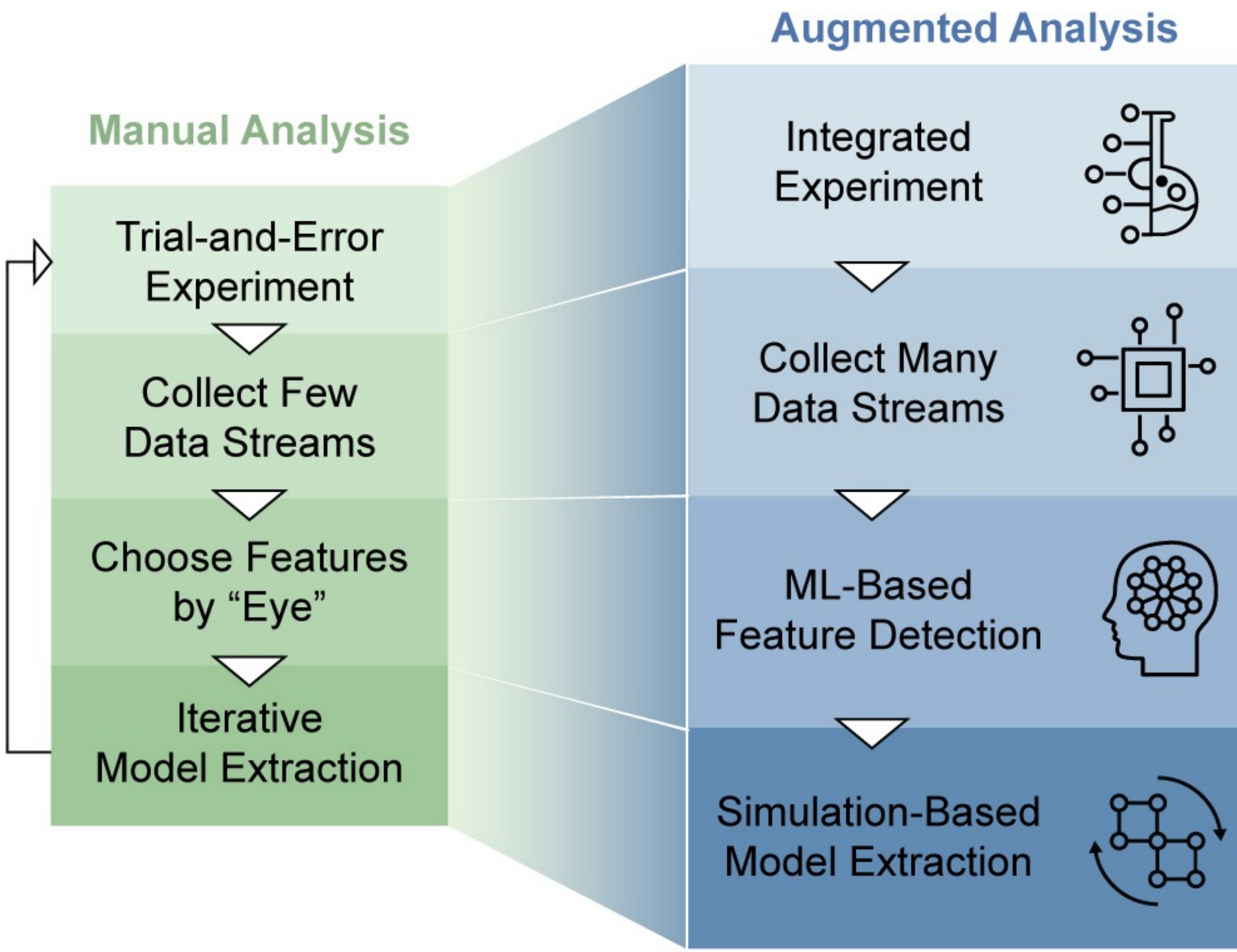
Scientific exploration is increasingly distributed & data-intensive

PNNL

- Domain science uses *workflows*:
  - Loose composition of different apps/tasks
    - motivated by productivity
  - Potentially different programming models
  - Data sources are distributed
  - Intensive use of memory, storage, networks
    - storage the means for task composition





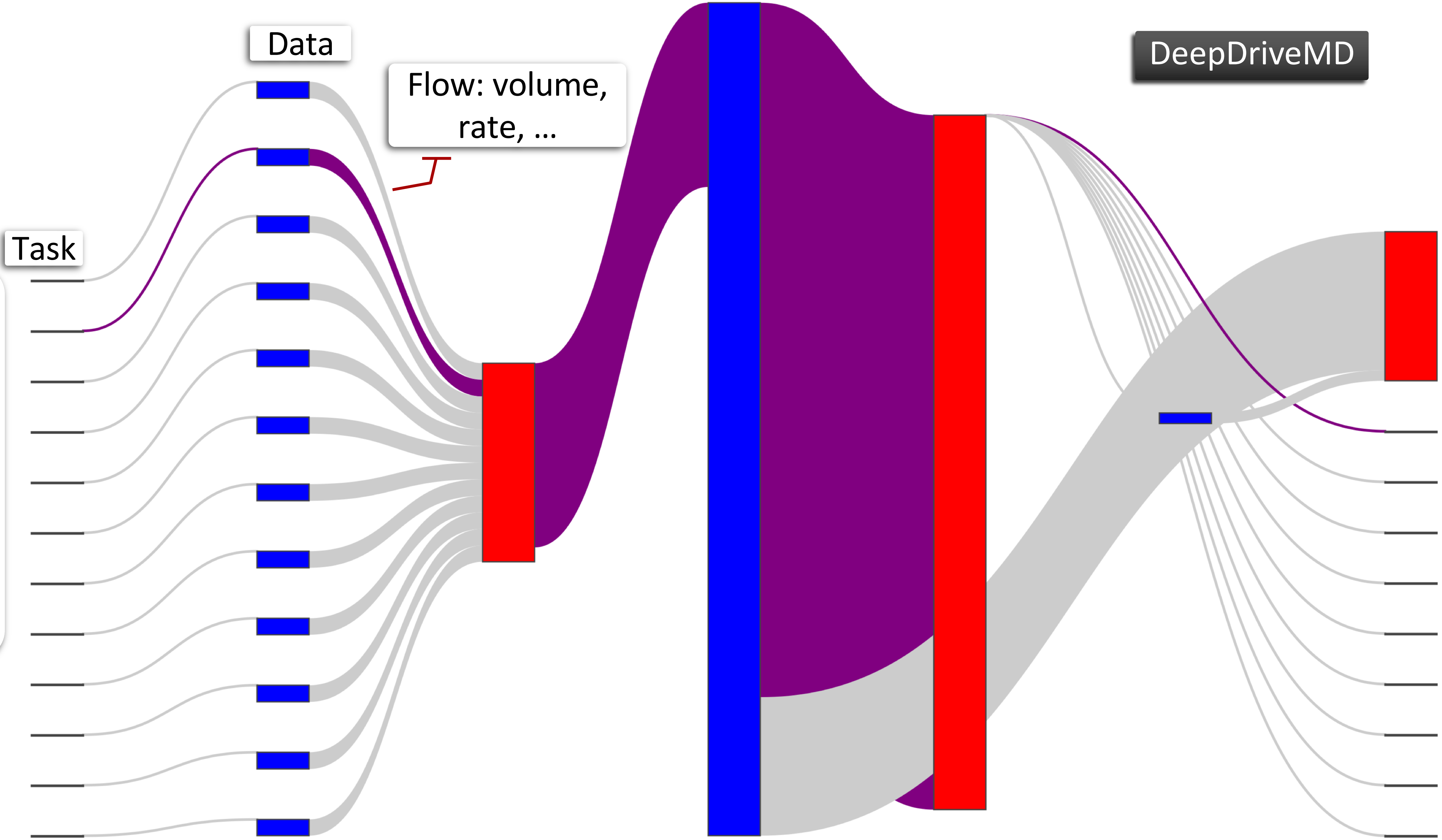


Data Flow Lifecycles: Runtime data & flow lifecycles

PNNL

Flow: left→right

Data Task



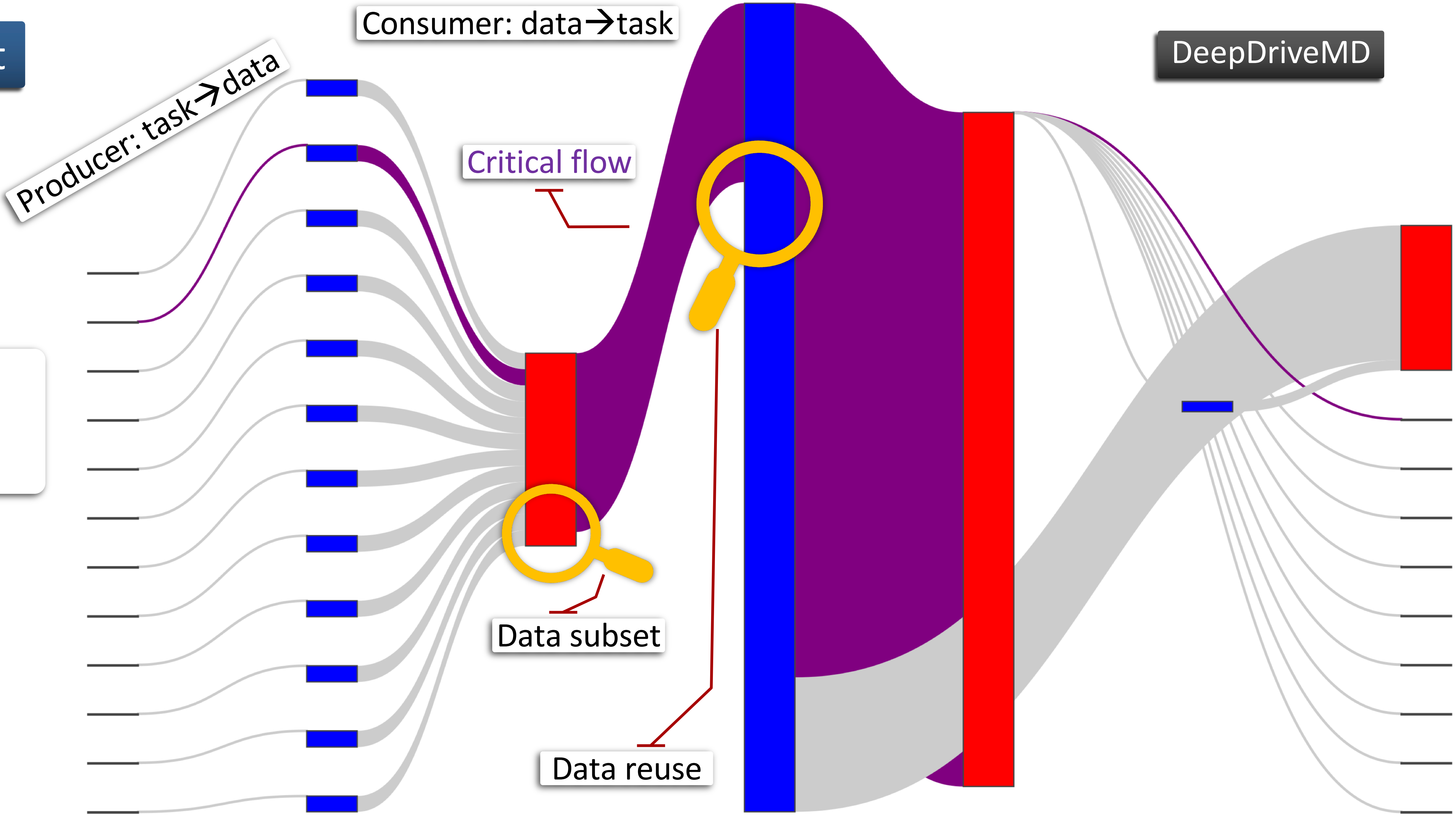
- Sankey diagrams: represent *flow*
- Vertices: Data & Task
- Flow: Edges
- Vertices and edges associated with many dynamic properties (affects rendering)

Data Flow Lifecycles: Runtime data & flow lifecycles

PNNL

Flow: left→right

Data Task



- Entities & Patterns (many more)

- Opportunity analysis… (not shown)