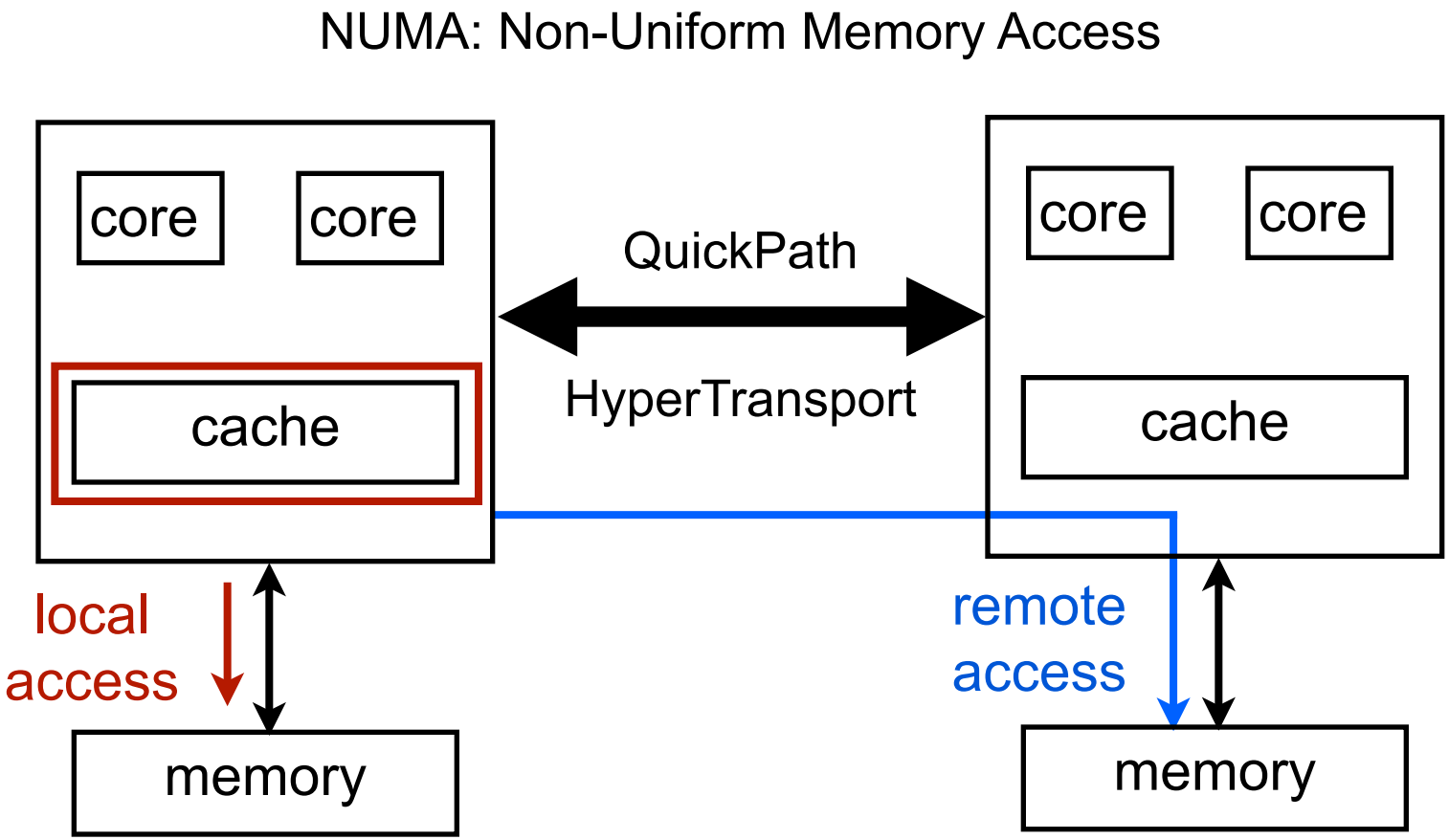
Leveraging Hardware Address Sampling

Beyond Data Collection and Attribution

Xu Liu

Department of Computer Science
College of William and Mary
xl10@cs.wm.edu

Motivation: Memory is the Bottleneck



State of the Arts

simulation methods

deep insights

weaknesses:

- 2-5x overhead
- not real machines

low overhead with deep insights

deep insights with low overhead

measurement methods

low overhead

Hardware Address Sampling

- Features of address sampling
  - sample memory-related events (memory accesses, NUMA events)
  - capture effective addresses
  - record precise IP of sampled instructions or events
- Support in modern processors
  - AMD Opteron 10h and above: instruction-based sampling (IBS)
  - IBM POWER 5 and above: marked event sampling (MRK)
  - Intel Itanium 2: data event address register sampling (DEAR)
  - Intel Pentium 4 and above: precise event based sampling (PEBS)
  - Intel Nehalem and above: PEBS with load latency (PEBS-LL)
- Efficient memory measurement (SC'13)
  - code-centric analysis
  - data-centric analysis

Code-centric vs. Data-centric

- Code-centric attribution
  - problematic code sections
    - instruction, loop, function

```
1: #pragma omp parallel for num_threads(4)
2: for (i = 0; i < n; i++) {
3:    for(j = 0; j < n; j++) {
4:       for(k = 0; k < n; k++) {
5:          A[i, j, k] = A[i, j, k,] + B[j, i, k] + C[k, j, i];
6:       }
7:    }
8: }
```

code-centric profiling

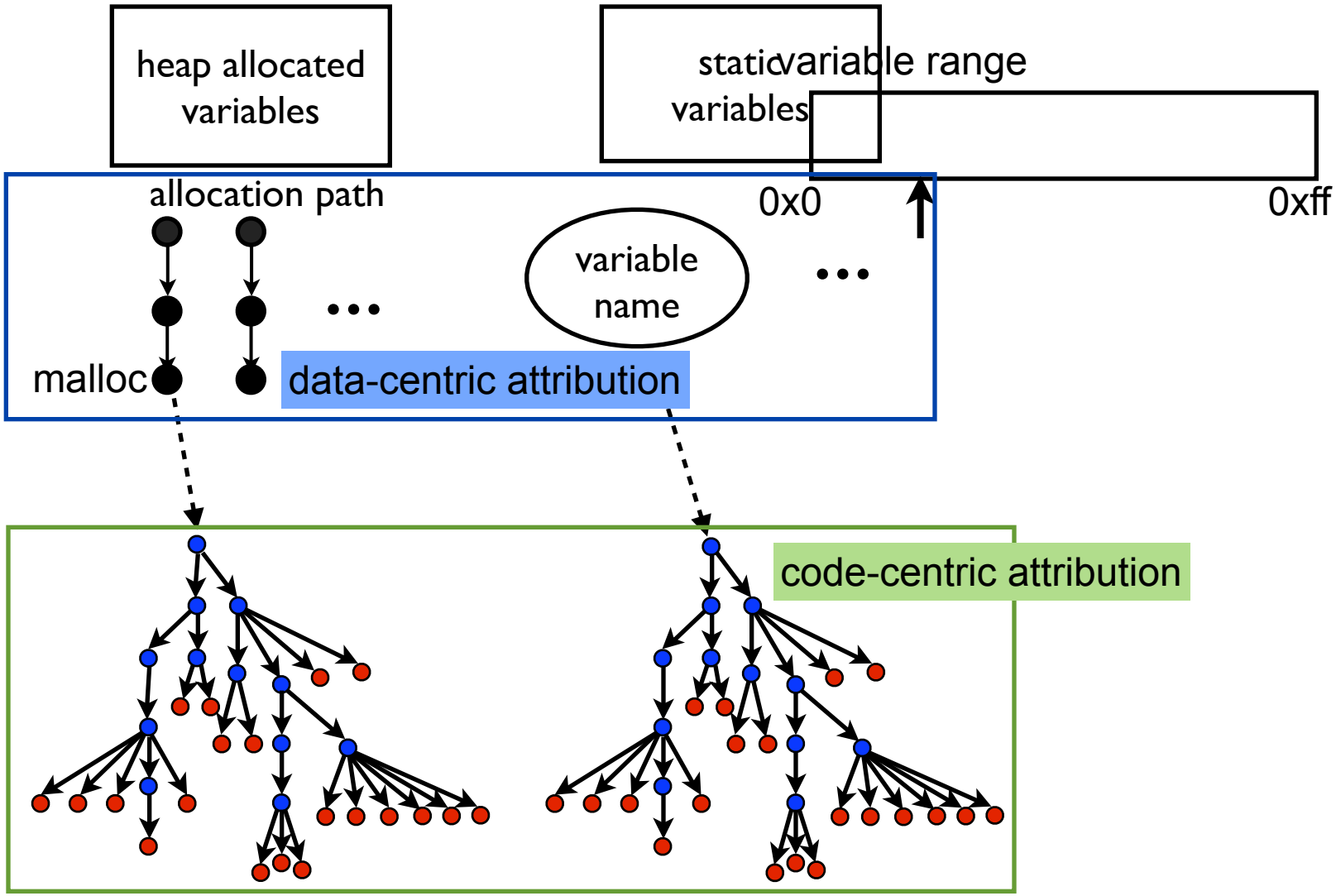
line 5: 100% latency

data-centric profiling

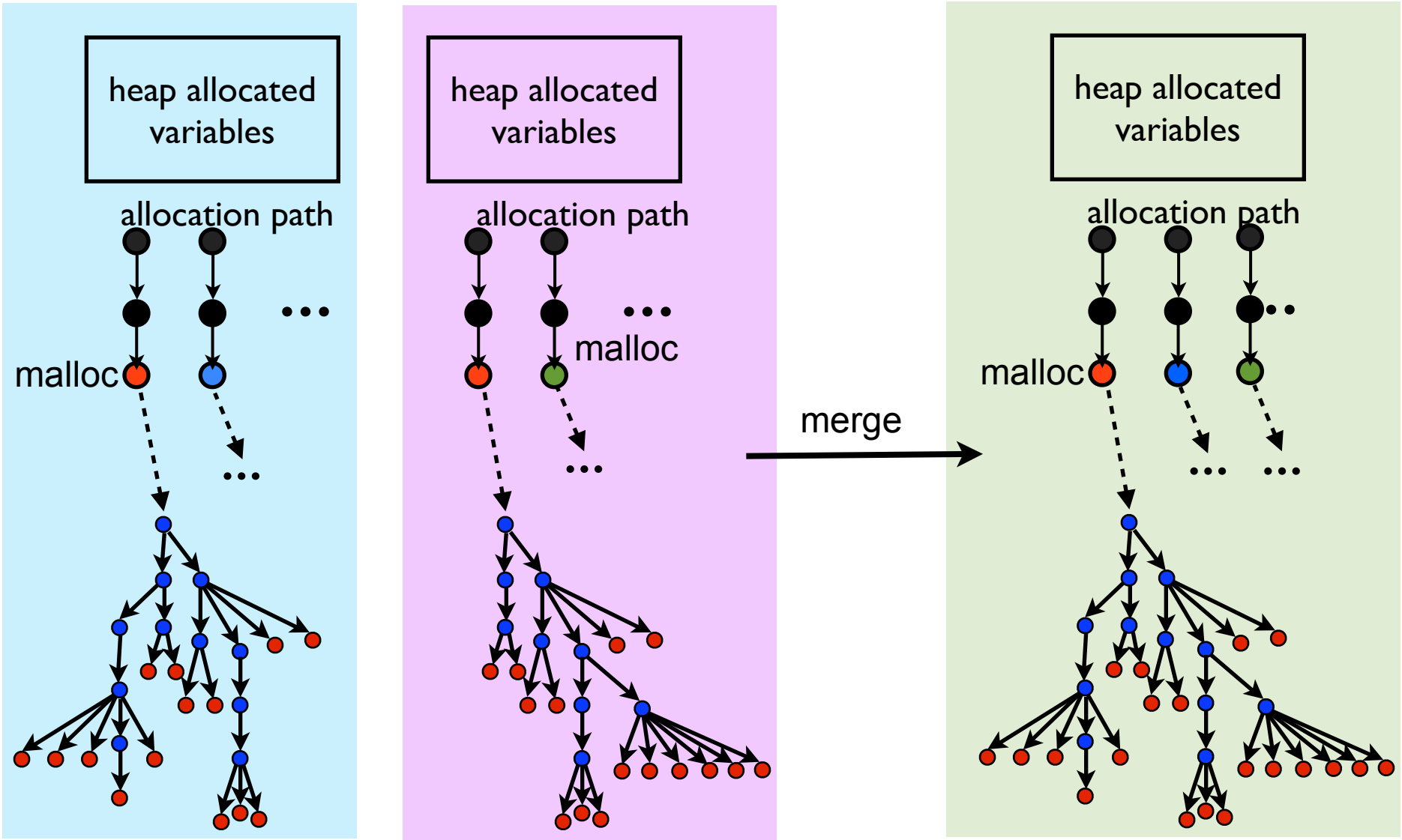
array A:
line 5: 1% latency
array B
line 5: 10% latency
array C
line 5: 89% latency

Combining code-centric and data-centric attribution provides additional insight

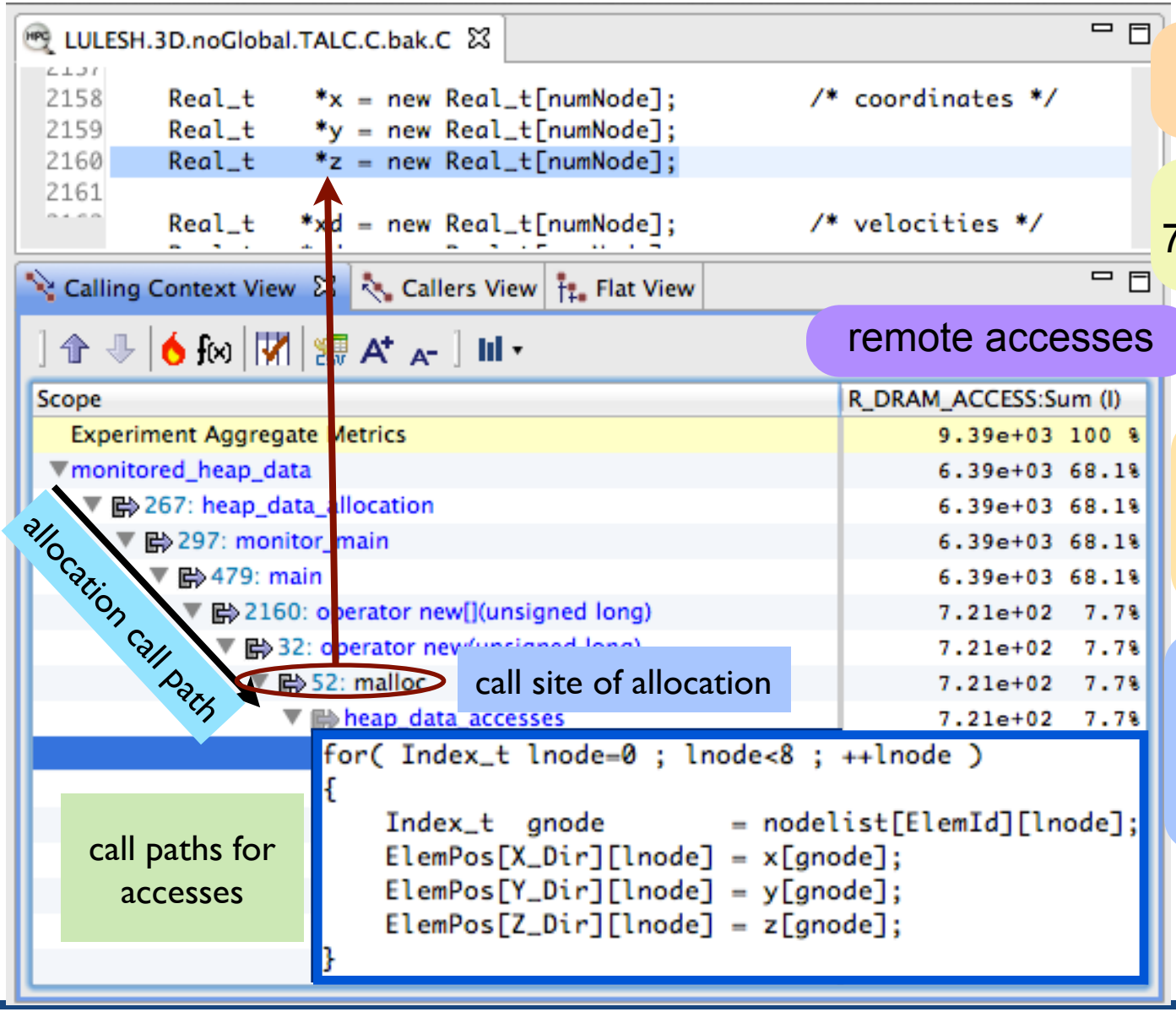
Attributing Samples



Aggregating Profiles



LULESH on Platform of 8 NUMA Domains



heap data:68% remote accesses

z accounts for 7.7% remote accesses

z is allocated in a NUMA domain but accessed by others

interleave pages of z across NUMA nodes 13% improvement in running time

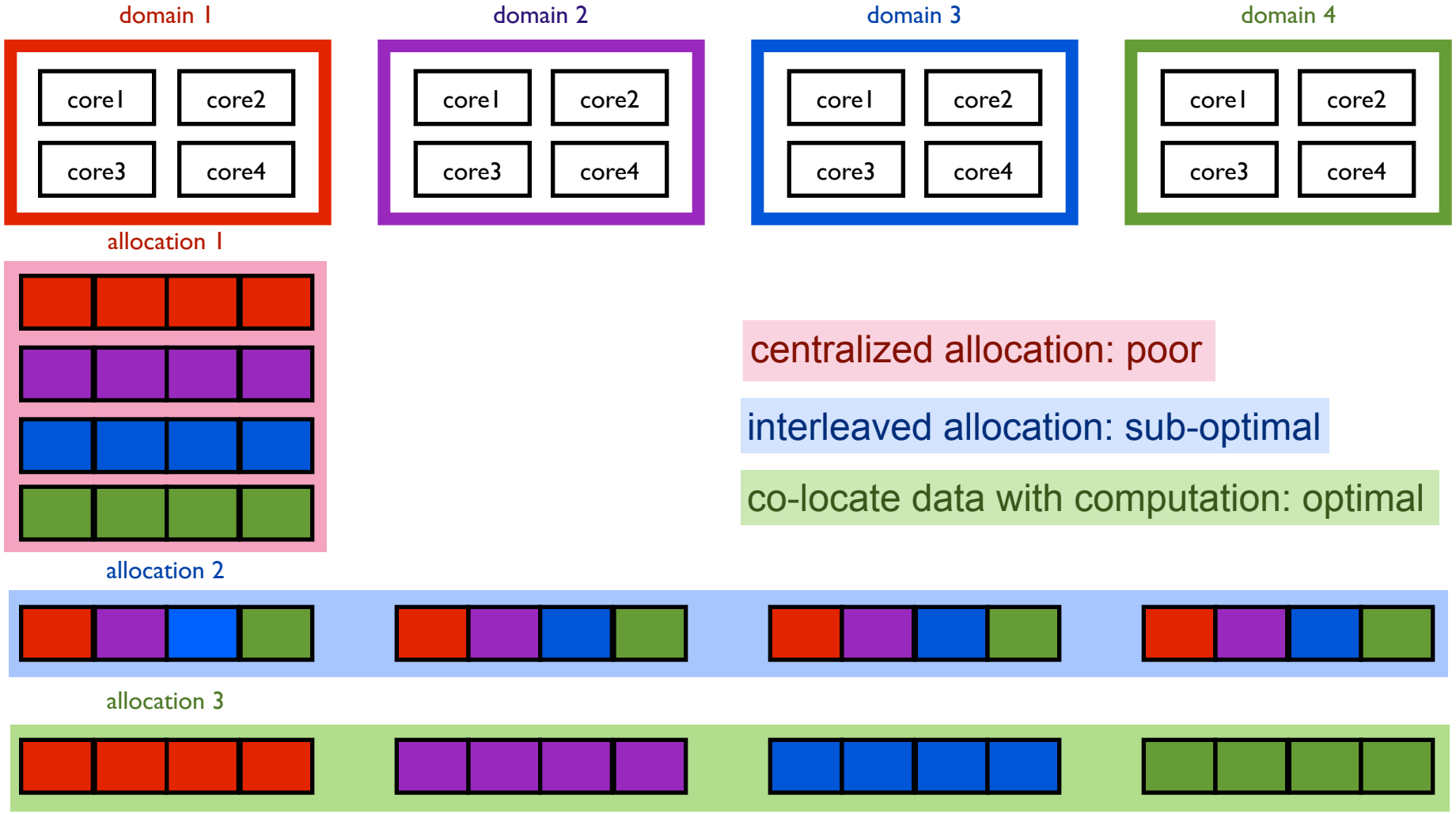
Existing Measurement is Inadequate

- Data collection + attribution ≠ optimal optimization
  - know problematic data objects but not know why
  - need more insights for optimization guidance
- Challenges for address sampling
  - very sparse memory access samples
  - not monitoring continuous memory accesses
- Opportunities for address sampling
  - effective addresses: analyze memory access patterns
  - data sources: understand where inefficiencies come from
  - latency: derive new latency metrics to quantify inefficiencies.

Beyond Data Collection and Attribution

- Published work
  - analyzing NUMA bottlenecks (PPoPP'14)
  - guiding array regrouping for better locality (PACT'14)
  - identifying memory scaling issues (SC'15)

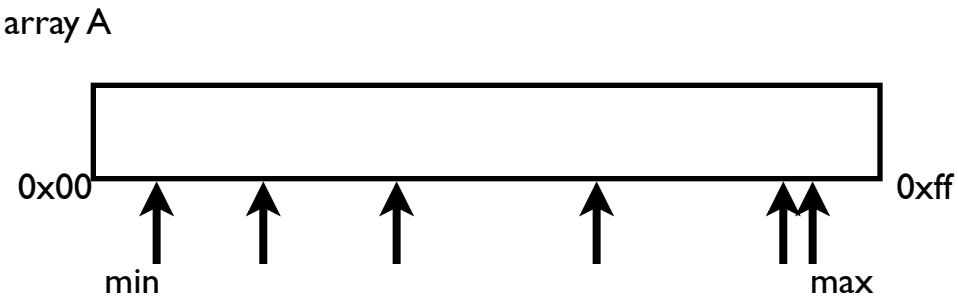
Interleaved Allocation is NOT Always Best



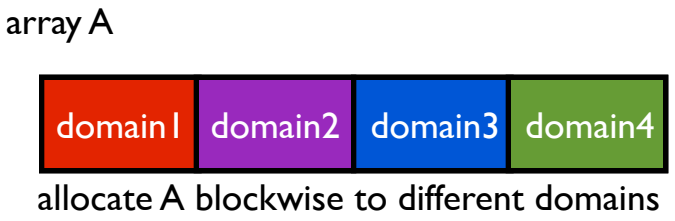
Goal: identify the best data distribution for a program

Memory Access Pattern Analysis

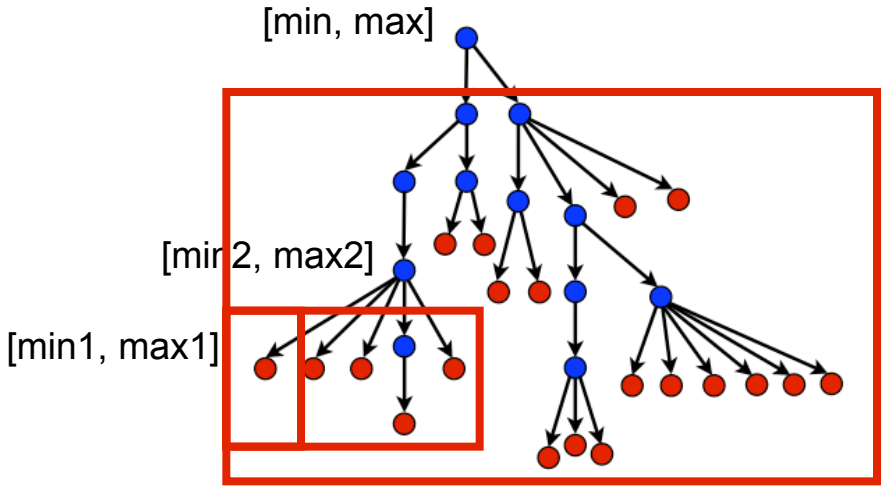
- Online data collection

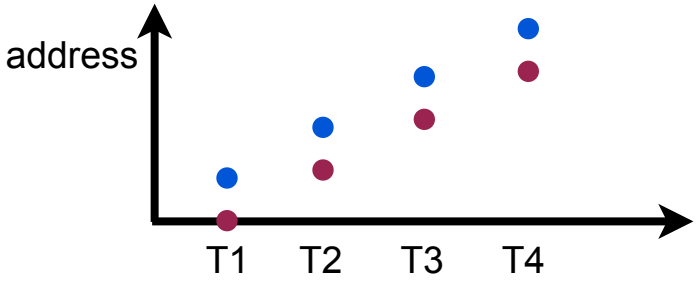


[min, max] per sampled memory access



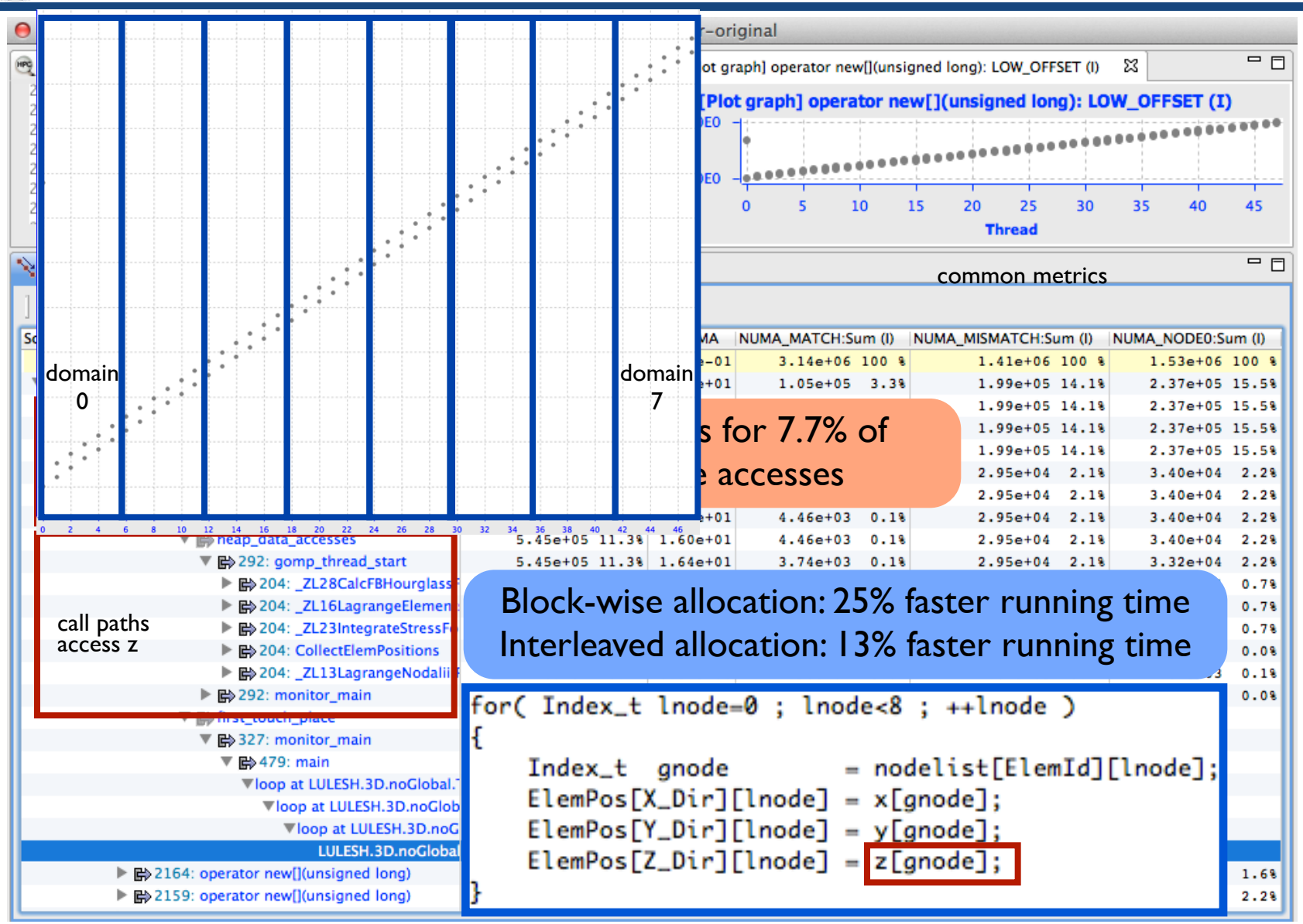
allocate A blockwise to different domains





balanced allocation + maximum locality

LULESH on Platform of 8 NUMA Domains



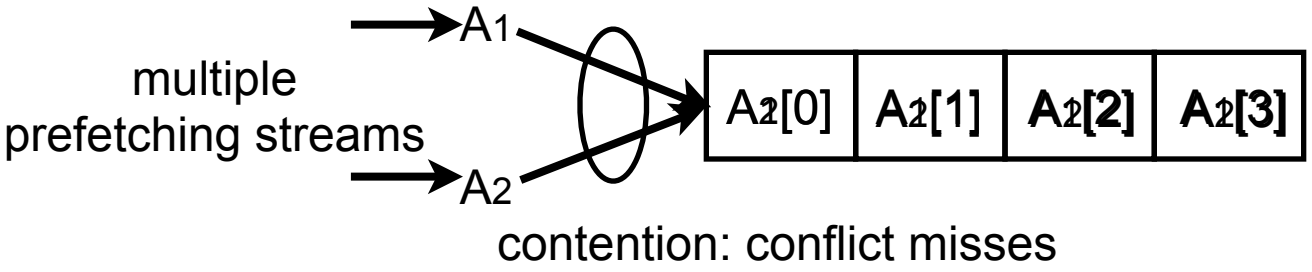
Beyond Data Collection and Attribution

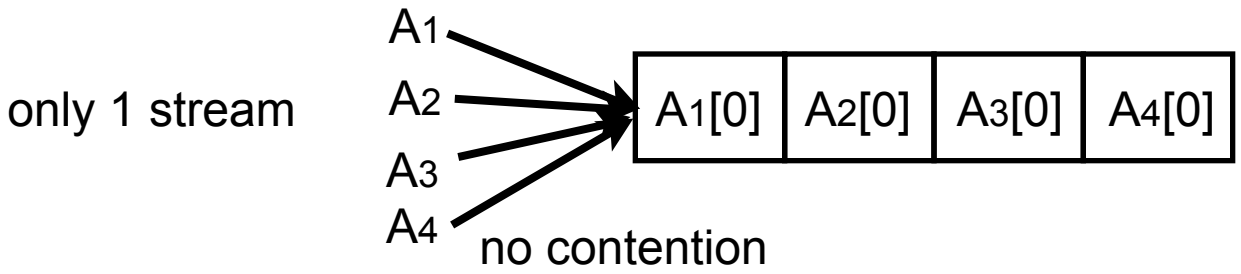
- Published work
  - analyzing NUMA bottlenecks (PPoPP'14)
  - guiding array regrouping for better locality (PACT'14)
  - identifying memory scaling issues (SC'15)

Array Regrouping

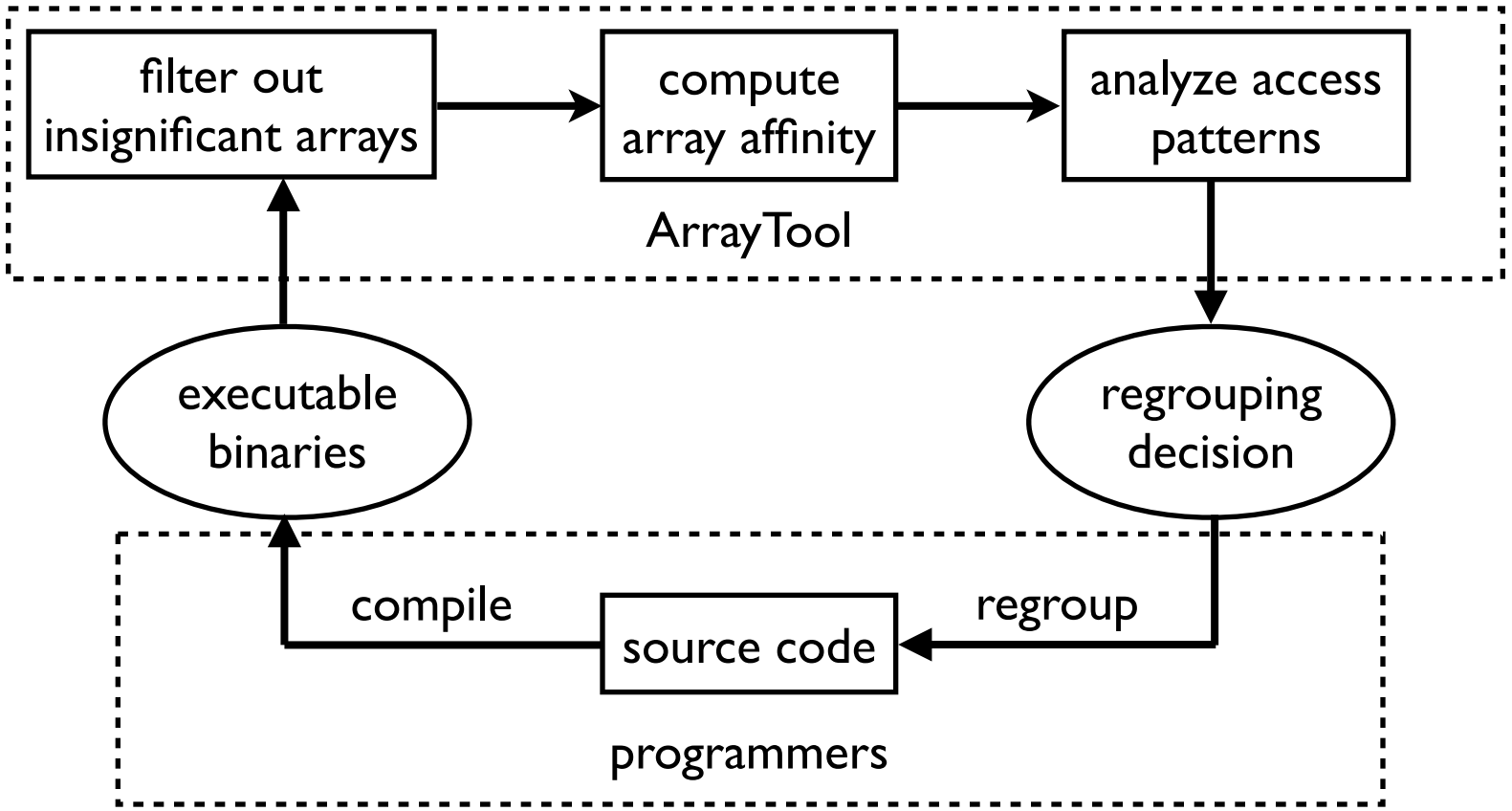
```
for (i = 0; i < N; i++)
    B[i] = A1[i] + A2[i] + ... + An[i];
```

```
for (i = 0; i < N; i++)
    B[i] = Arr[i].A1 + Arr[i].A2 + ... + Arr[i].An;
```





Workflow


filter out insignificant arrays
compute array affinity
analyze access patterns
ArrayTool
executable binaries
regrouping decision
compile
source code
regroup
programmers

Filter out Insignificant Arrays

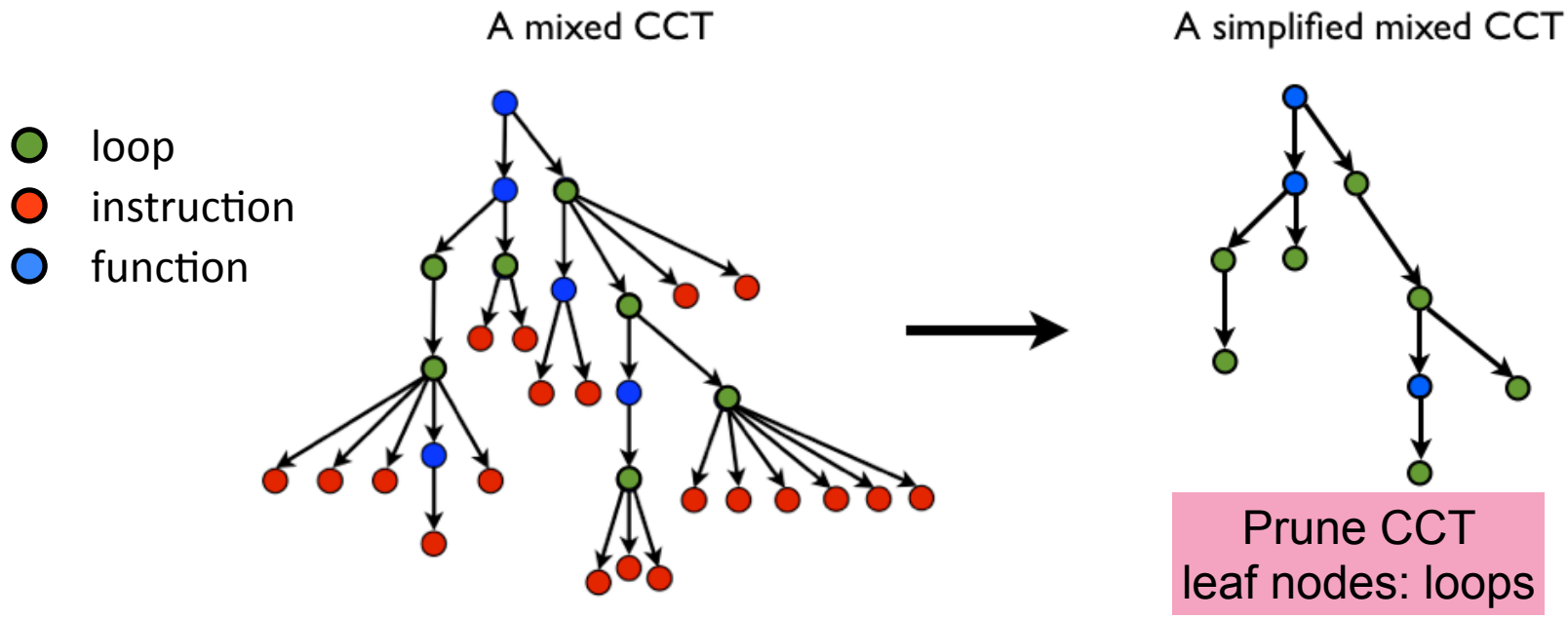
latency

```
0.0%  I = (float *)malloc( size_I * sizeof(float) );
12.4% J = (float *)malloc( size_I * sizeof(float) );
4.9%  c  = (float *)malloc(sizeof(float)* size_I) ;

0.0%  iN = (int *)malloc(sizeof(unsigned int*) * rows) ;
0.0%  iS = (int *)malloc(sizeof(unsigned int*) * rows) ;
0.1%  jW = (int *)malloc(sizeof(unsigned int*) * cols) ;
1.5%  jE = (int *)malloc(sizeof(unsigned int*) * cols) ;

2.7%  dN = (float *)malloc(sizeof(float)* size_I) ;
3.8%  dS = (float *)malloc(sizeof(float)* size_I) ;
5.1%  dW = (float *)malloc(sizeof(float)* size_I) ;
5.6%  dE = (float *)malloc(sizeof(float)* size_I) ;
```

SRAD from Rodinia

Latency-based Array Affinity

loops in their calling contexts

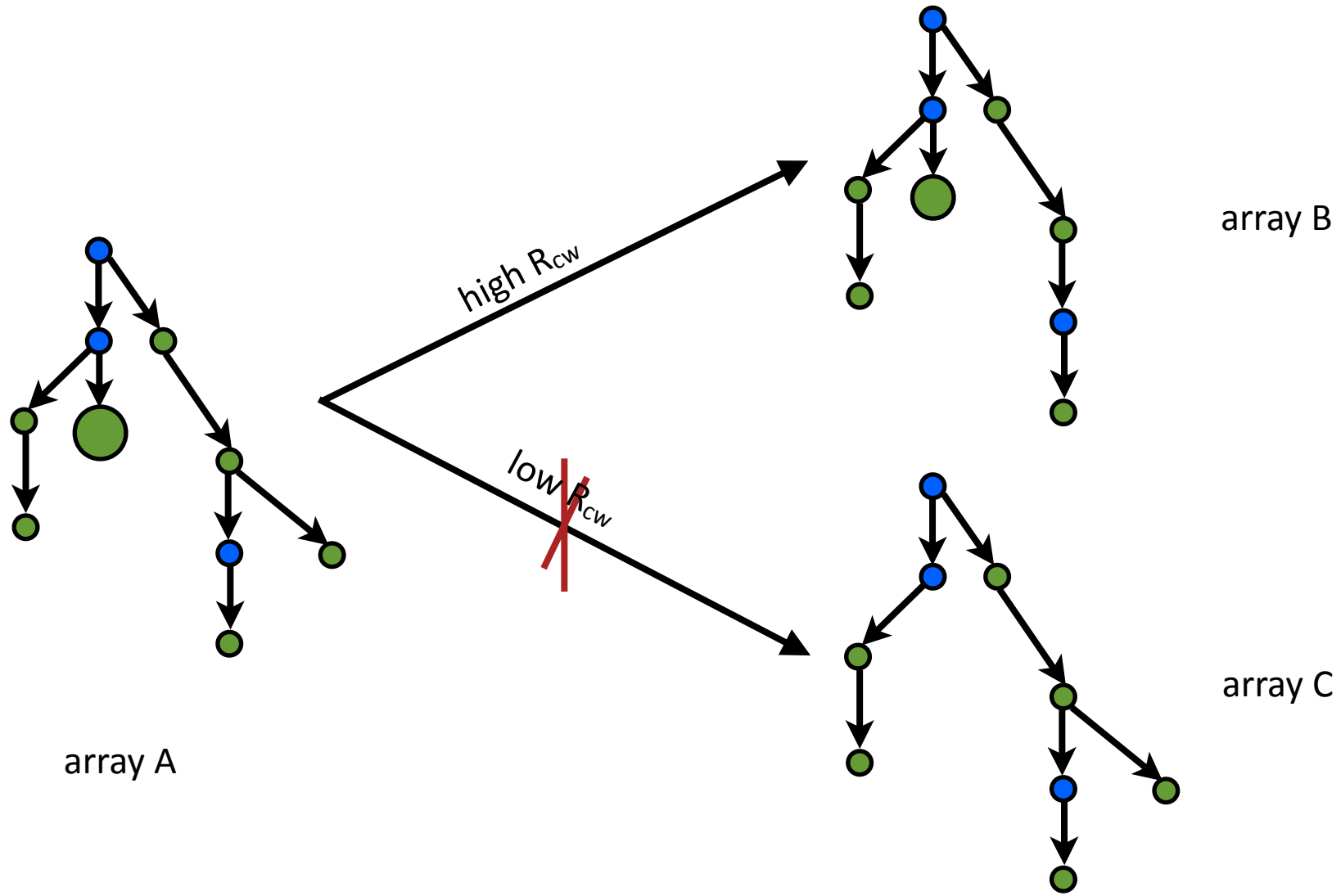


$$R_{cw} = 1 - \frac{W_{uncommon}}{W_{total}}$$

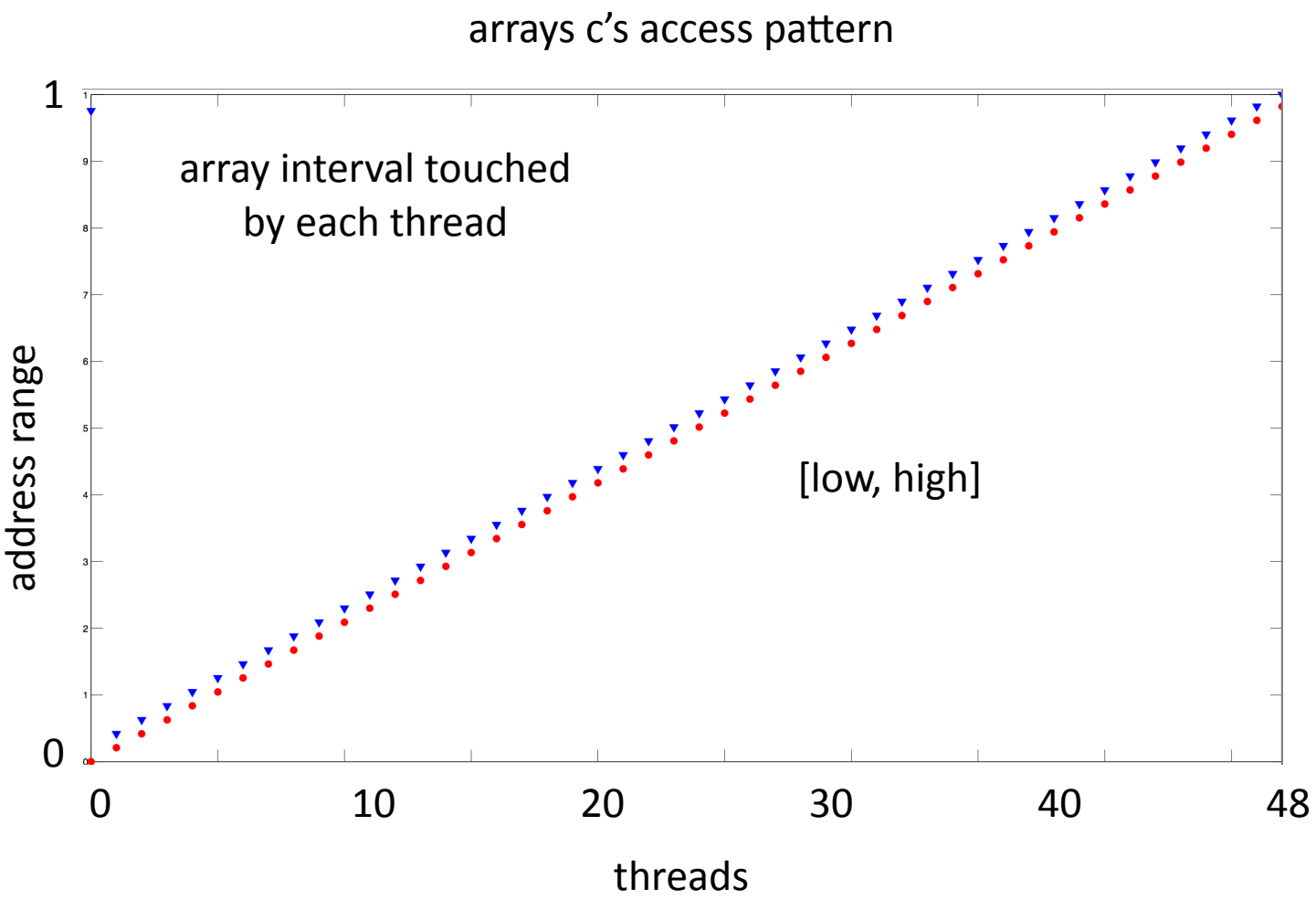
aggregate latency of uncommon loops

aggregate latency of all loops

An Example



Access Pattern Analysis



dN, dS, dE, dW have the same pattern

Regrouping Results of SRAD

```
#pragma omp parallel for
for(int i=0; i<rows; i++) {
  for(int j=0; j<cols; j++) {
      k = i*cols+j;
      cN = c[k];
      cS = c[iS[i]*cols+j];
      cW = c[k];
      cE = c[i*cols+jE[j]];
      D = cN*dN[k]+cS*dS[k]+cW*dW[k]+cE*dE[k];
      J[k] = J[k]+0.25*lambda*D;
  }
}
```

only regroup dN, dS, dW, dE → ↑1.47x

regroup dN, dS, dW, dE and c → ↑1.89x

Beyond Data Collection and Attribution

- Published work
  - analyzing NUMA bottlenecks (PPoPP'14)
  - guiding array regrouping for better locality (PACT'14)
  - identifying memory scaling issues (SC'15)

Scaling Losses in Memory Hierarchies

- Memory contentions hurt scalability: cache/bandwidth contention
  - which data objects contribute to the most scaling losses
  - which memory layers incur the most scaling losses
- Methods
  - decompose latency according to data objects and memory layers
    - data-centric analysis with data source information
  - differential analysis supported by HPCToolkit
    - compare profiles between different runs

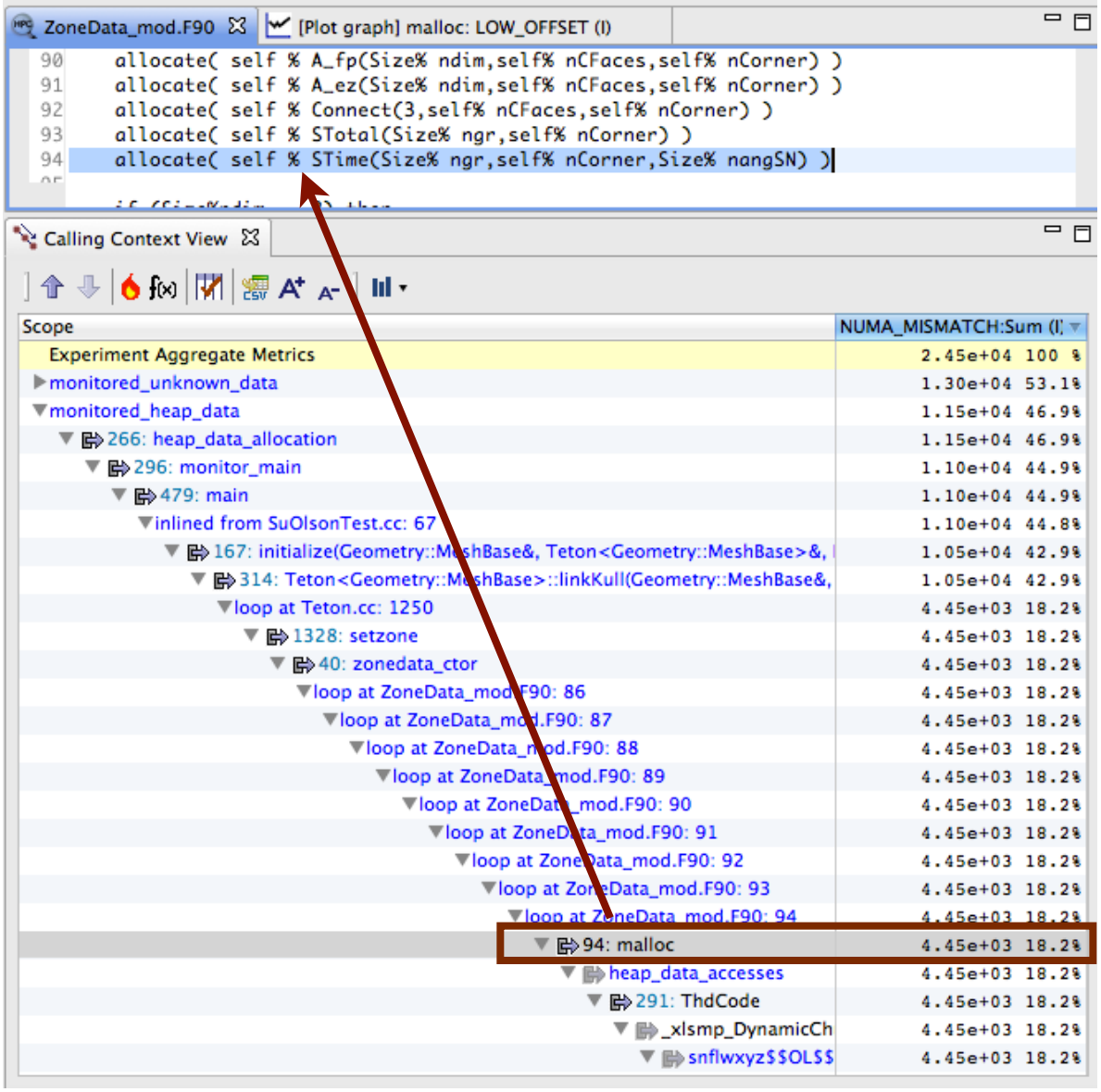
more details in SC'15

Conclusions and Future Work

- Hardware address sampling
  - widely supported in modern architectures
  - powerful in monitoring memory behaviors
  - more analysis of the samples provides more performance insights
- On-going work
  - structure splitting
  - locality optimization between SMT threads
  - cache line false sharing
  - automatic page migration for NUMA architectures
- Future directions of address sampling
  - comparing different address sampling mechanisms
  - analyzing new performance issues
    - heterogeneous memory: 3D stack memory

Backup Slides

UMT2013 on Quad-socket POWER7 Node



sample off-chip accesses

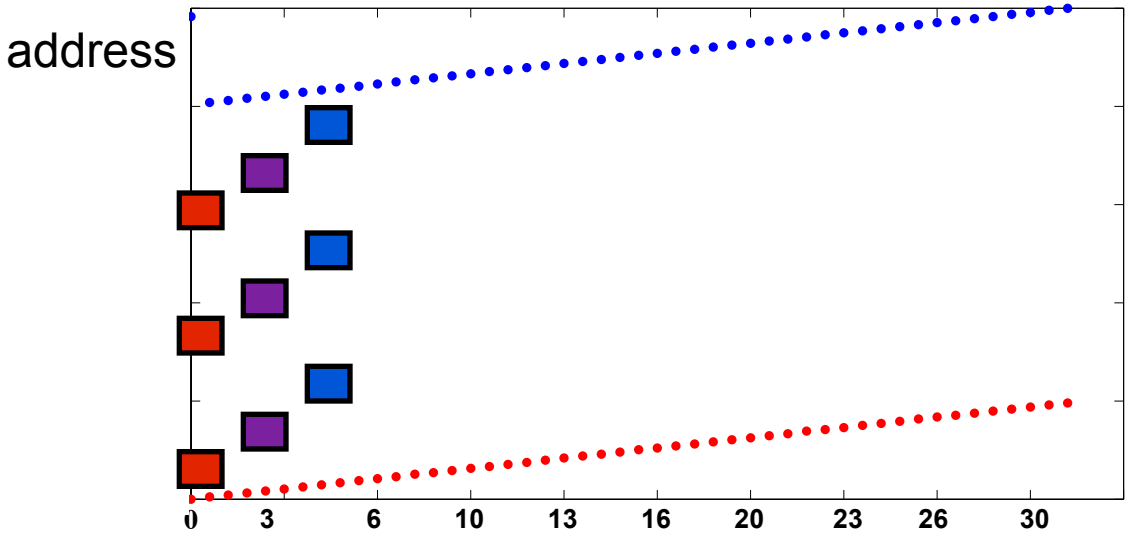
self%STime

18.2% of remote accesses

allocated in one domain accessed by everyone

Optimize *self%STime* for UMT2013

address-centric analysis for *self%STime*



self%STime's address space

optimization: let each thread initialize its own data

result: all threads have data locally -- 7% faster