Featherlight Reuse-distance Measurement

Qingsen Wang, Xu Liu

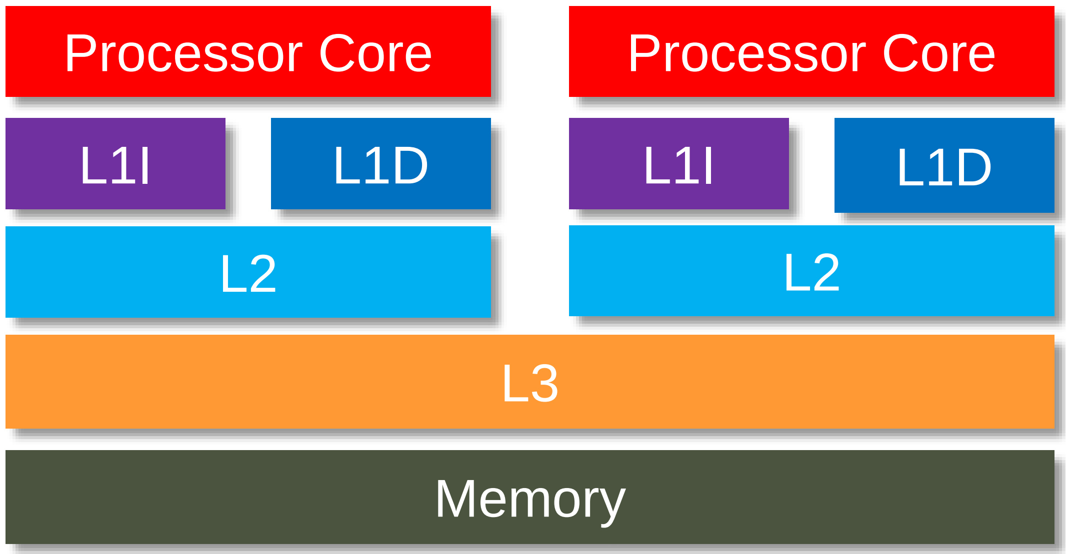
College of William & Mary

Milind Chabbi

Scalable Machines Research

Run on Modern Memory Hierarchy

- Complex memory hierarchy

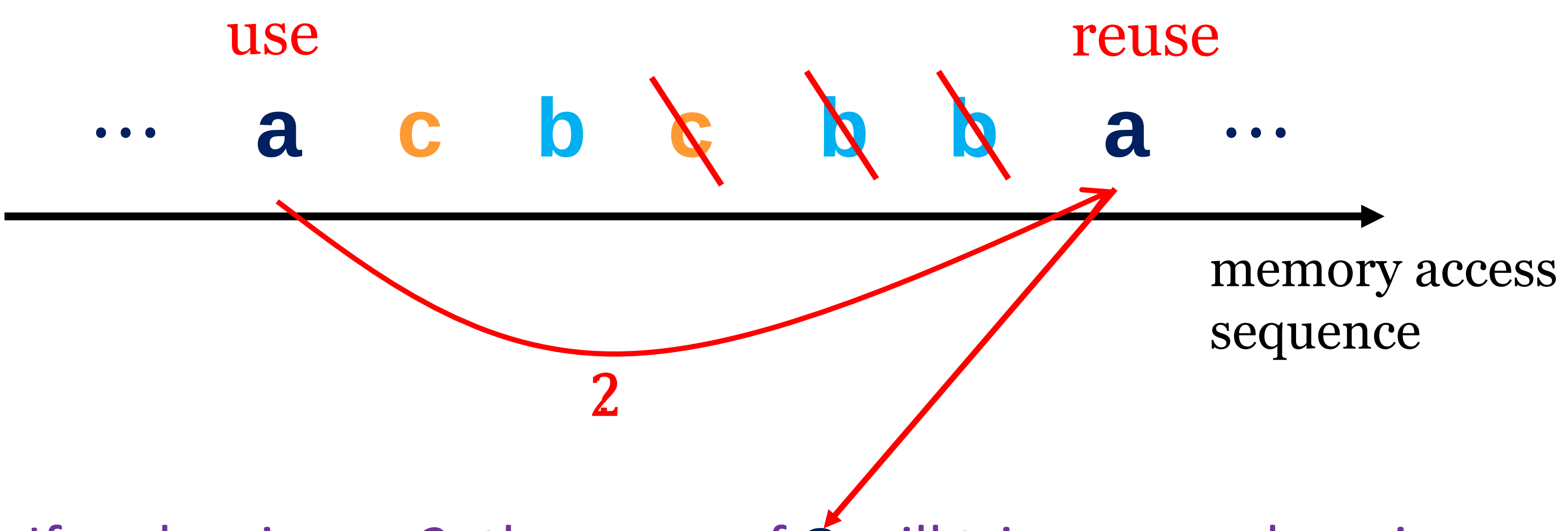


- The working set size of programs keeps growing

Managing data locality becomes more and more important to memory/cache performance

Quantify Data Locality

- Reuse distance
  - Stack reuse distance, stack distance
  - The number of distinct memory locations between two consecutive uses (of the same memory location)



If cache size <= 2, the reuse of **a** will trigger a cache miss.

  - Highly related to cache miss ratio
  - Focus on reuse distance of the whole program

Quantify Data Locality

- Why reuse distance?
  - Software metric independent from hardware
  - Performance prediction and analysis
  - Cache simulation
  - Program phase prediction
  - Code optimization
  - …

Profile Reuse Distance

- Profiling reuse distance of the whole program is costly
  - Exhaustive instrumentation tool: 100X~1000X slowdown

- Our solution – RDX
  - A sampling-based profiler to measure reuse distance of the whole program aided by hardware
  - No instrumentation
  - No recompilation
  - Low overhead: ~5%(time), ~7%(memory)
  - High accuracy: >90%

RDX – Design Overview

Sample memory access address

Measure time distance of the sampled address

Time distance → reuse distance

RDX – Sample Memory Access

Sample memory access address

Measure time distance of the sampled address

Time distance → reuse distance

RDX – Sample Memory Access

- Performance Monitor Units (PMU)
  - Available in commodity CPUs
  - Monitor hardware events

    e.g. CPU cycles, instructions, L1D cache misses
  - Count the occurrence of an event
  - Interrupt the program when the monitored event's occurrence reaches the expected number

    i.e., PMU sample

- RDX counts/samples LOAD and STORE events
  - Each PMU sample comes with the corresponding memory reference location (e.g., effective address from Intel PEBS)

RDX – Sample Memory Access

Sample memory access address

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

Measure time distance of the sampled address

Time distance → reuse distance

RDX – Measure Time Distance

Sample memory access address

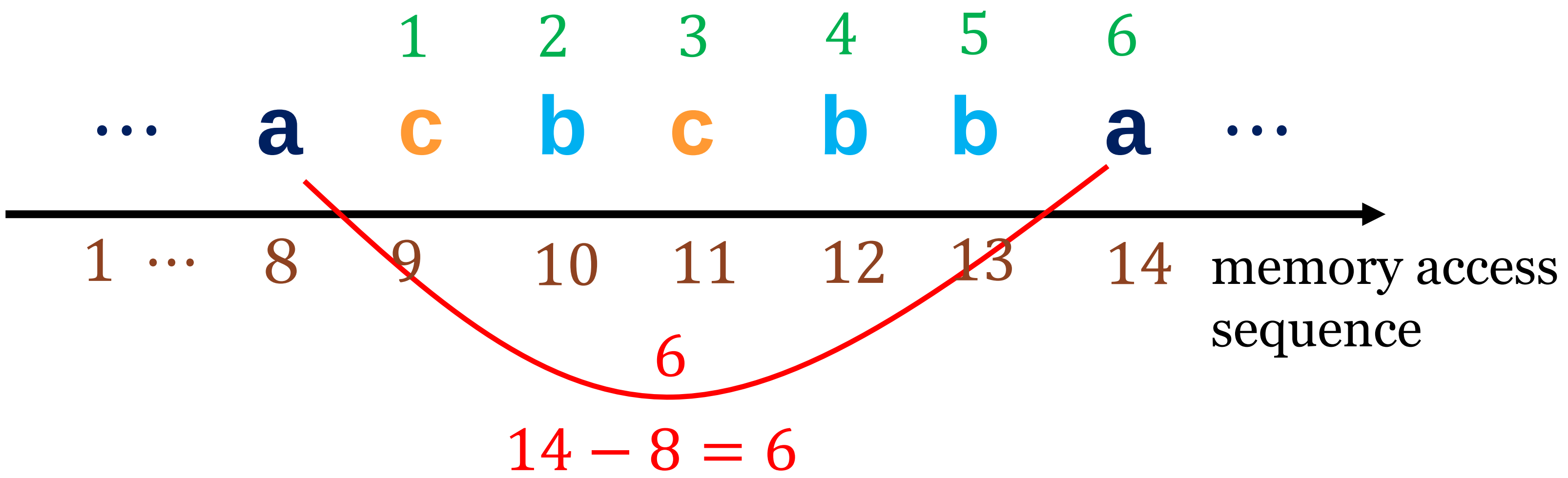
- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

Measure time distance of the sampled address

Time distance → reuse distance

RDX – Measure Time Distance

- Time distance
  - The number of memory accesses since last use



- Why time distance?
  - No need to maintain history to remove duplicates
  - Cheaper to measure than reuse distance.

RDX – Measure Time Distance

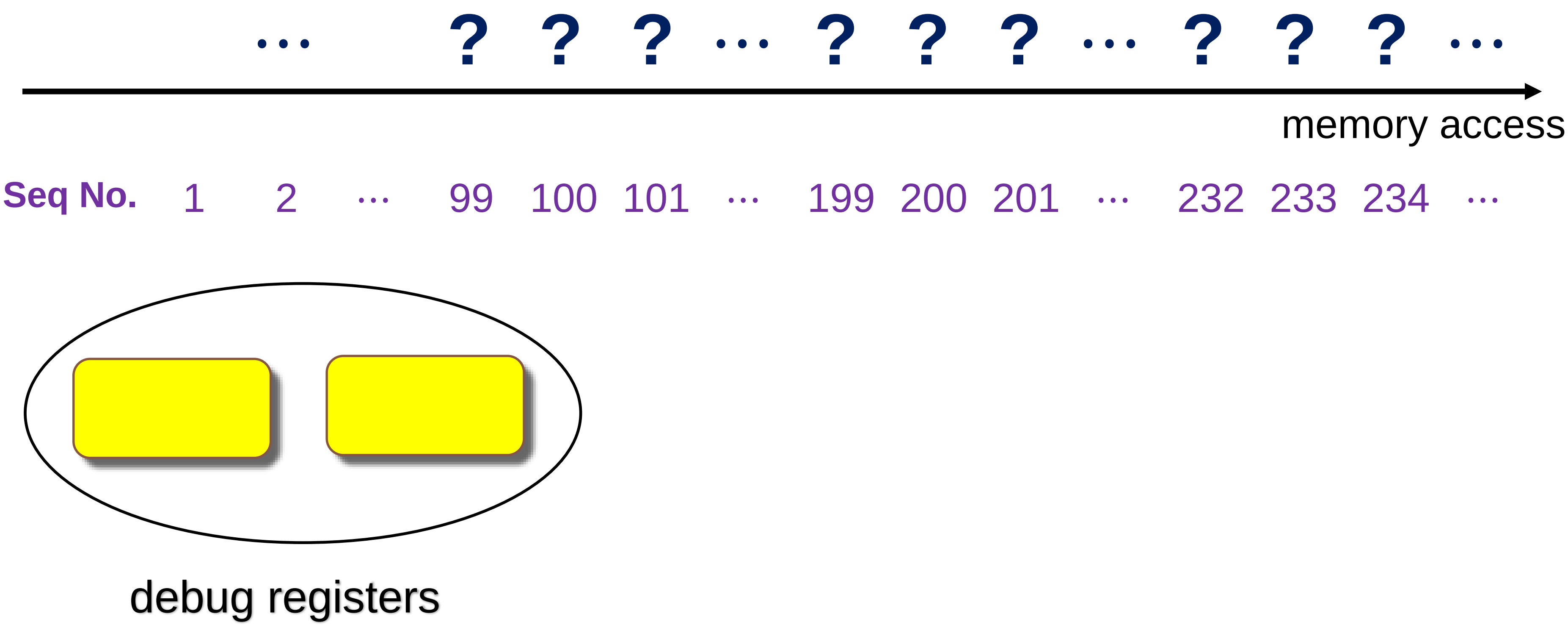
- Debug register
  - Available on most commodity CPUs
  - Subscribe

    Monitor a memory location
  - Trap

    Interrupt the program once the monitored memory location is accessed

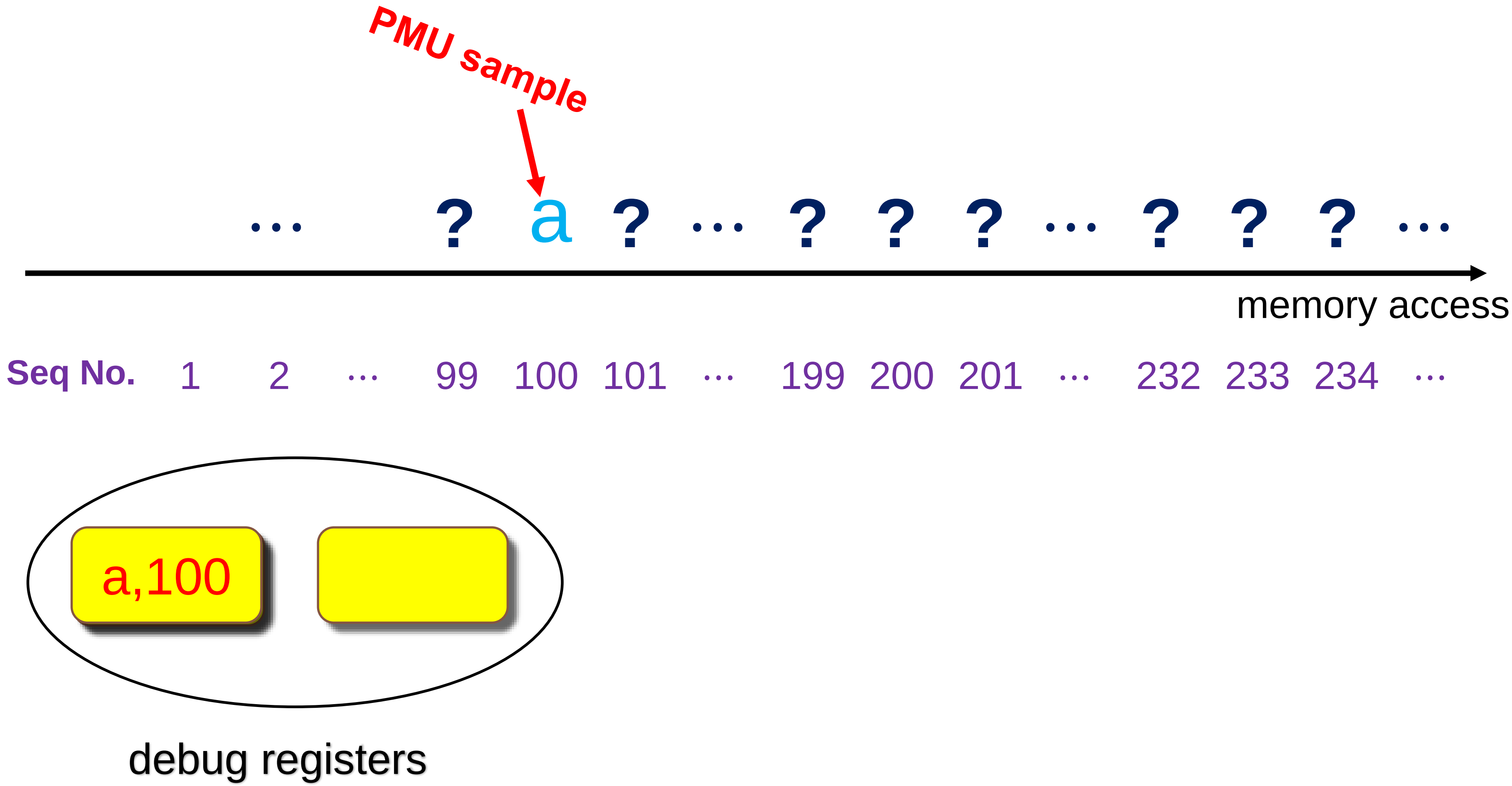
RDX – Measure Time Distance

- Use debug register to measure time distance
  - PMU samples every 100 memory references.



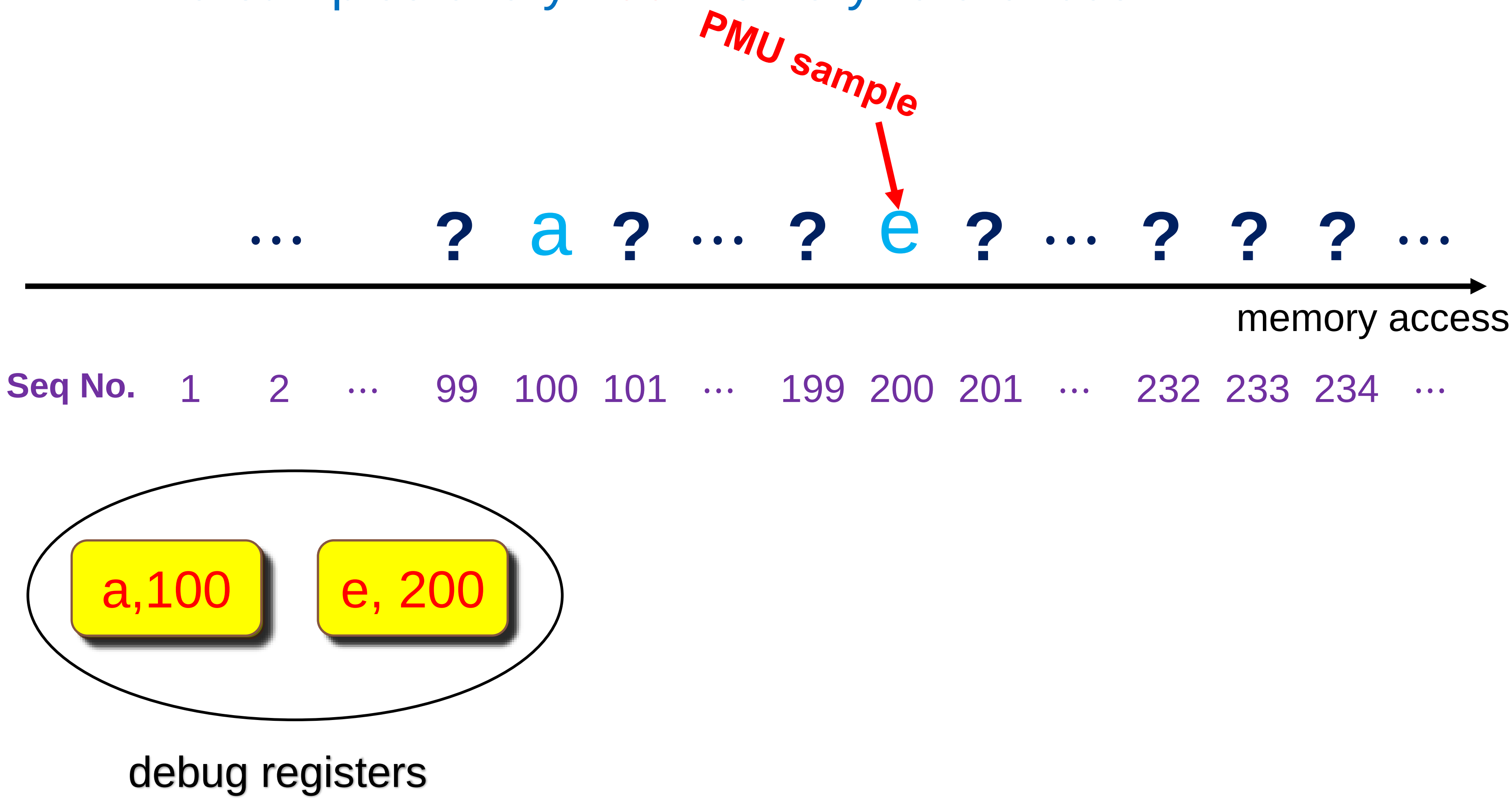
RDX – Measure Time Distance

- Use debug register to measure time distance
  - PMU samples every 100 memory references.



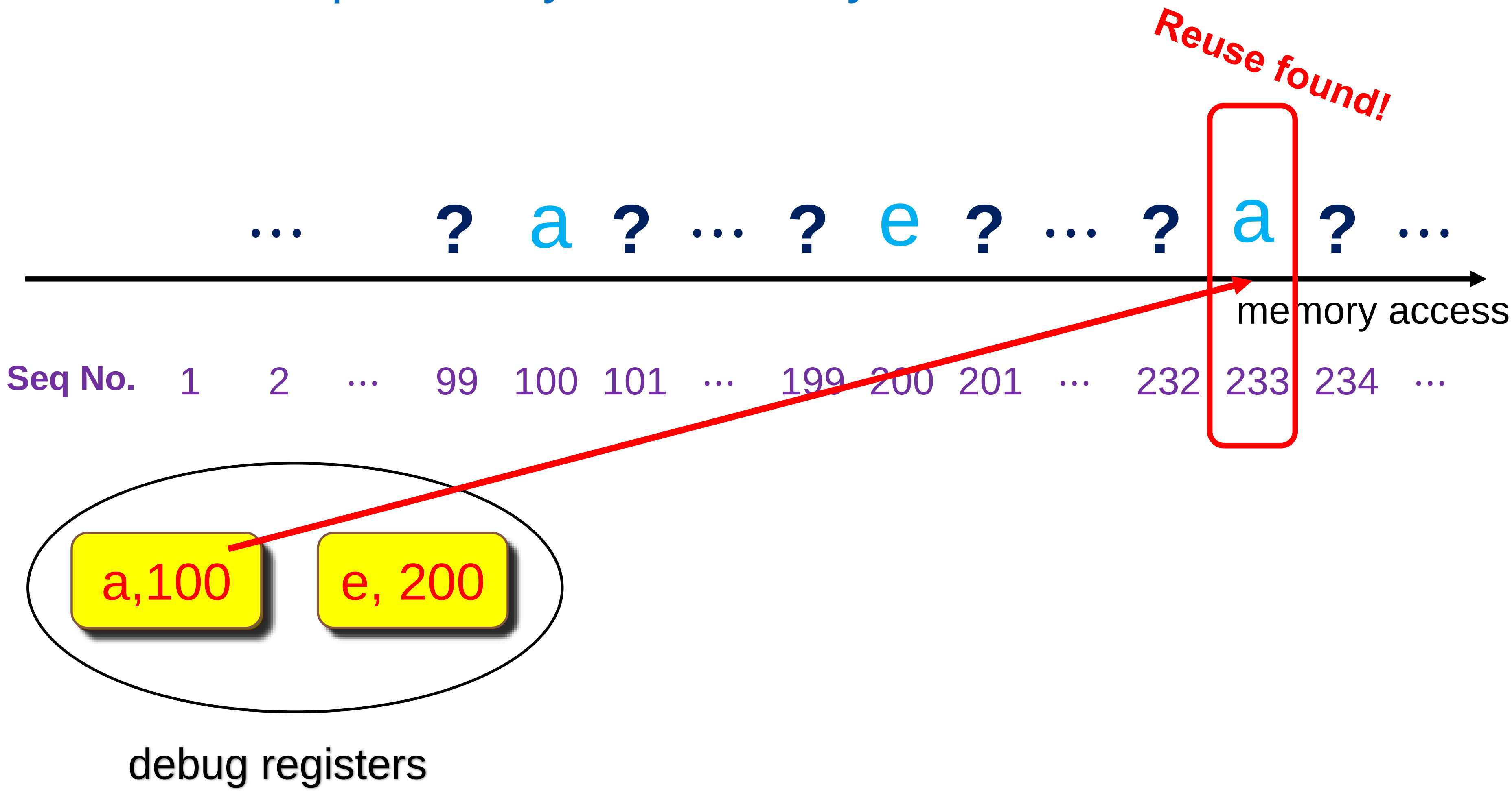
RDX – Measure Time Distance

- Use debug register to measure time distance
  - PMU samples every 100 memory references.



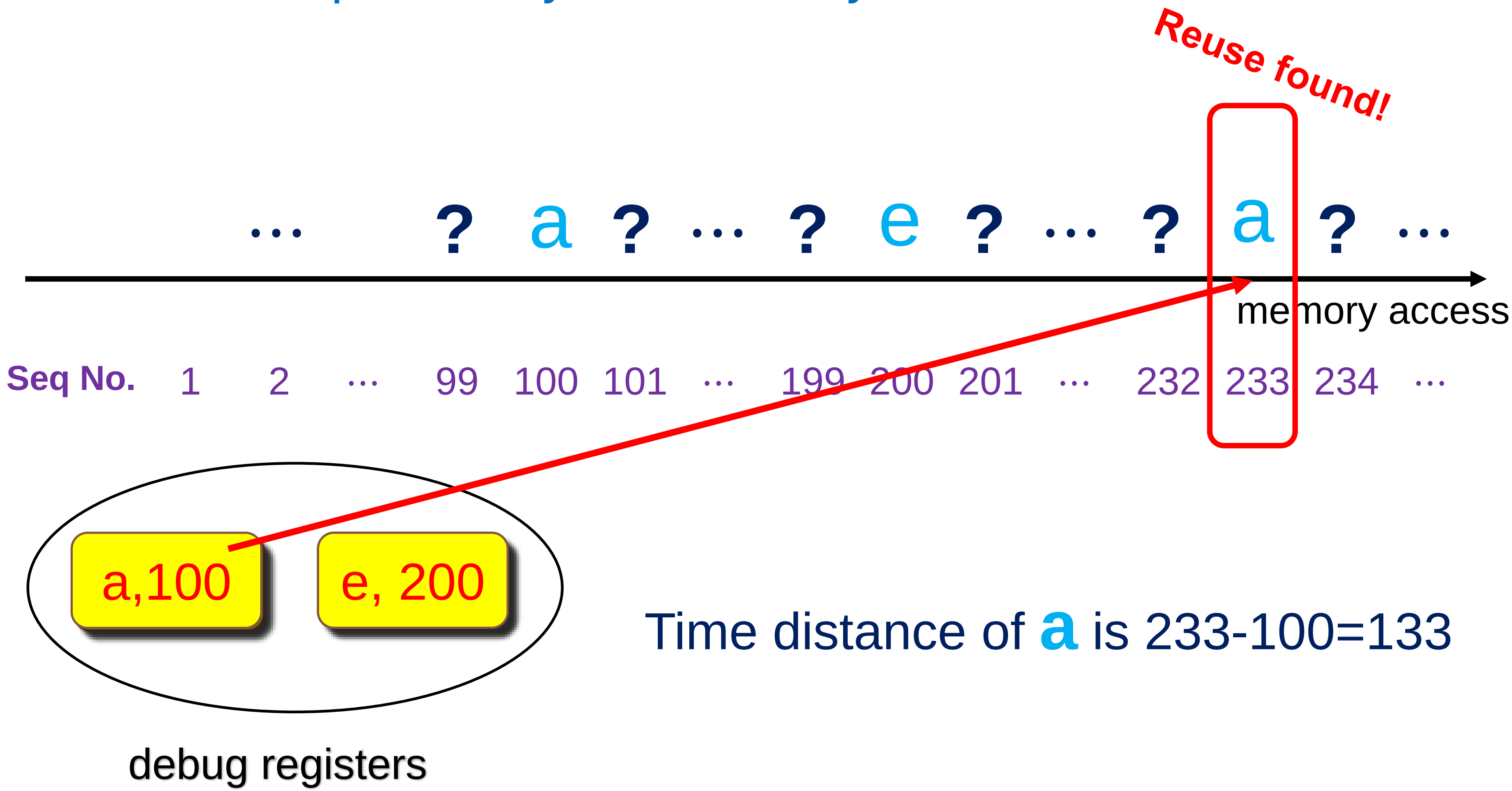
RDX – Measure Time Distance

- Use debug register to measure time distance
  - PMU samples every 100 memory references.



RDX – Measure Time Distance

- Use debug register to measure time distance
  - PMU samples every 100 memory references.



RDX – Measure Time Distance

Sample memory access address

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

Measure time distance of the sampled address

- Use debug registers to detect the reuse position of a memory location

Time distance → reuse distance

RDX – Time → Reuse

Sample memory access address

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

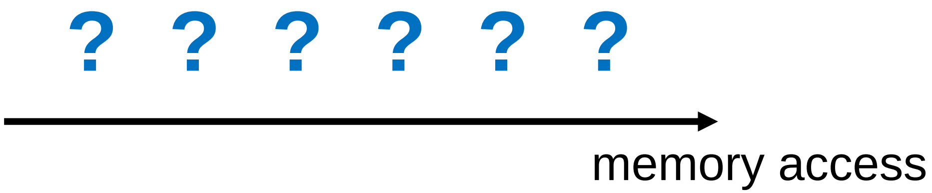
Measure time distance of the sampled address

- Use debug registers to detect the reuse position of a memory location

Time distance → reuse distance

RDX – Time → Reuse

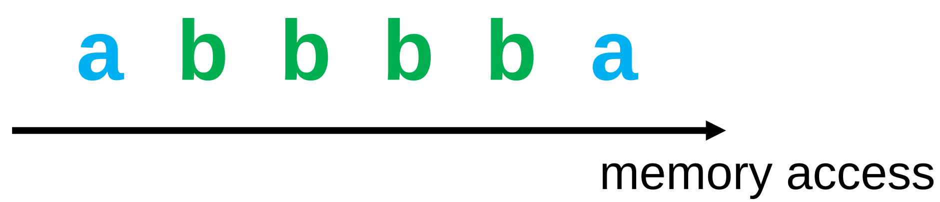
- How is time distance related to stack distance?



Time distance	Occurrence
1	3
5	1

RDX – Time → Reuse

- How is time distance related to stack distance?

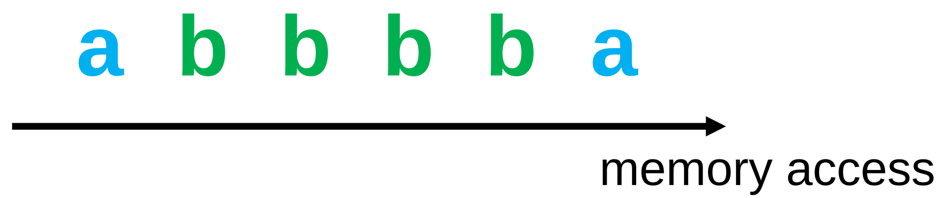


Time distance	Occurrence
1	3
5	1

Reuse distance	Occurrence
0	3
1	1

RDX – Time → Reuse

- How is time distance related to stack distance?



Time distance	Occurrence
1	3
5	1

Reuse distance	Occurrence
0	3
1	1

Not feasible to enumerate all the possibilities for real programs

RDX – Time → Reuse

- Statistically convert time distance to reuse distance

Locality Approximation Using Time (POPL'07)	
Assumption	A data element is accessed independently from others, which is a Bernoulli process.
Input	Time distance histogram, max working size
Output	Stack distance histogram

RDX – Time → Reuse

Sample memory access address

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

Measure time distance of the sampled address

- Use debug registers to detect the reuse position of a memory location

Time distance → reuse distance

- Each data location is accessed independently
- Statistically estimate reuse distance histogram from time distance

RDX – Review

Sample memory access address

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access

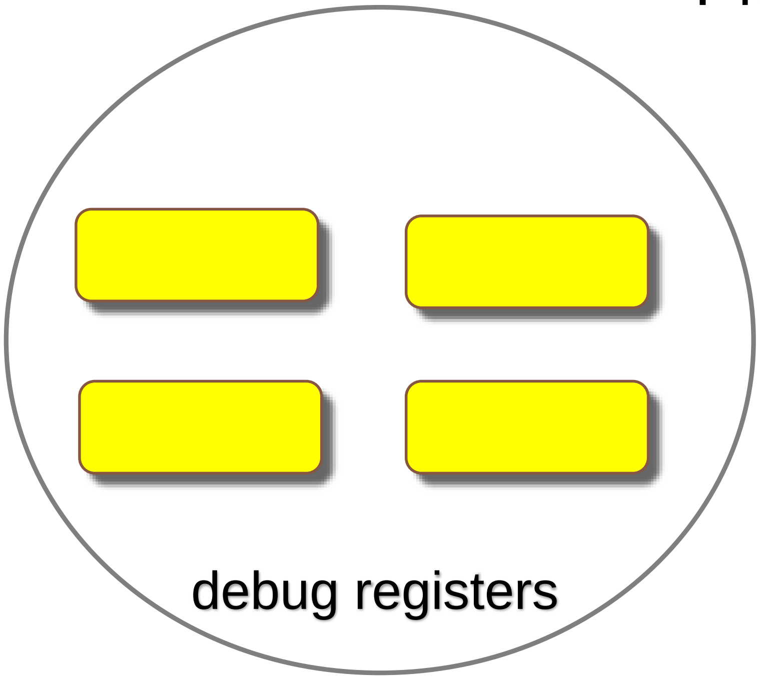
Measure time distance of the sampled address

- Use debug registers to detect the reuse position of a memory location

Time distance → reuse distance

- Each data location is accessed independently
- Statistically estimate reuse distance histogram from time distance

Challenge

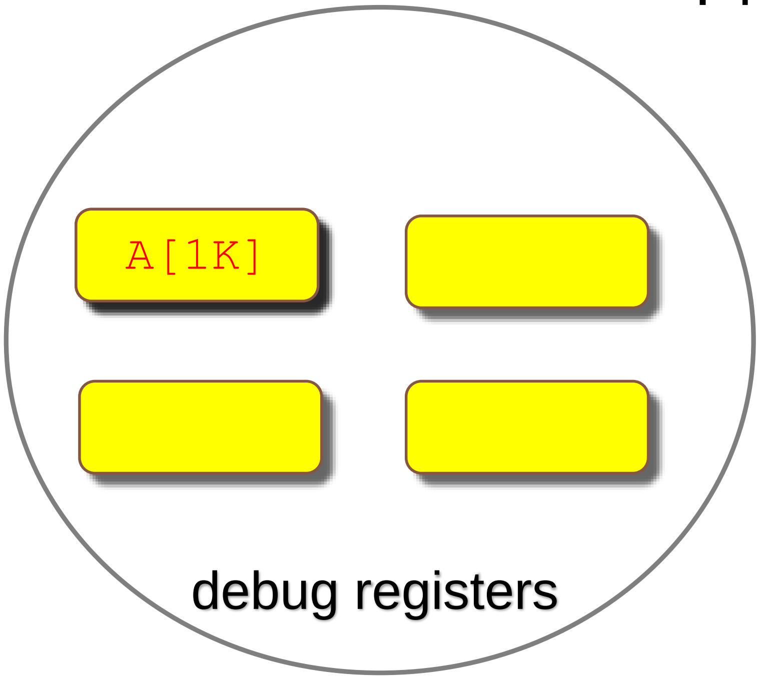


PMU samples every 1K memory stores

```
for (int i=1; i<=10K; i++){
   A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

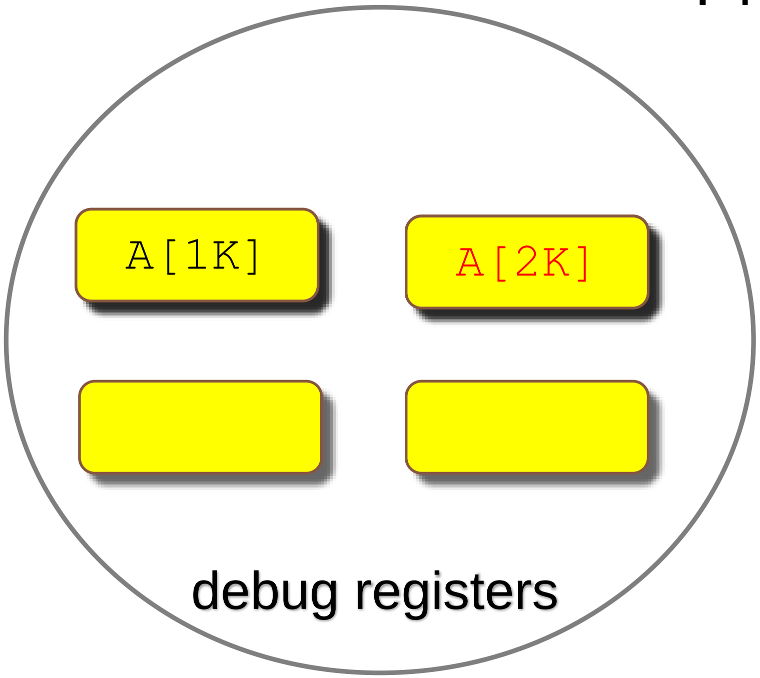
PMU samples every 1K memory stores




```
for (int i=1; i<=10K; i++){
   A[i] = 0;      i=1K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

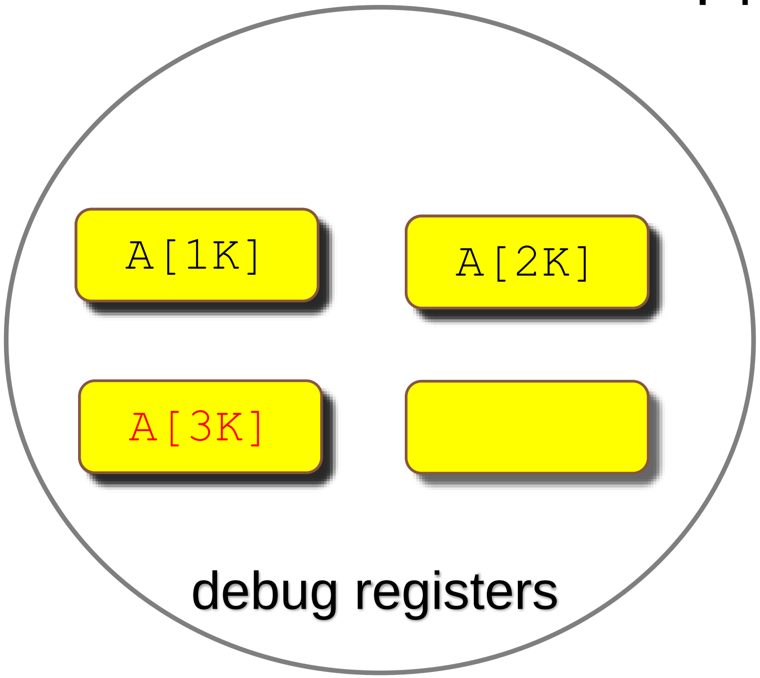
PMU samples every 1K memory stores




```
for (int i=1; i<=10K; i++){
   A[i] = 0;     i=2K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

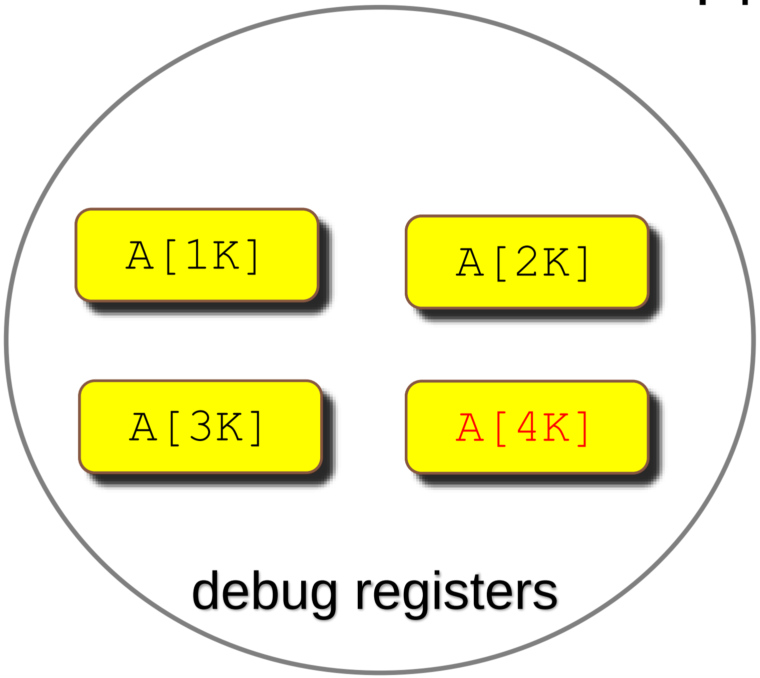
PMU samples every 1K memory stores




```
for (int i=1; i<=10K; i++){
   A[i] = 0;     i=3K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

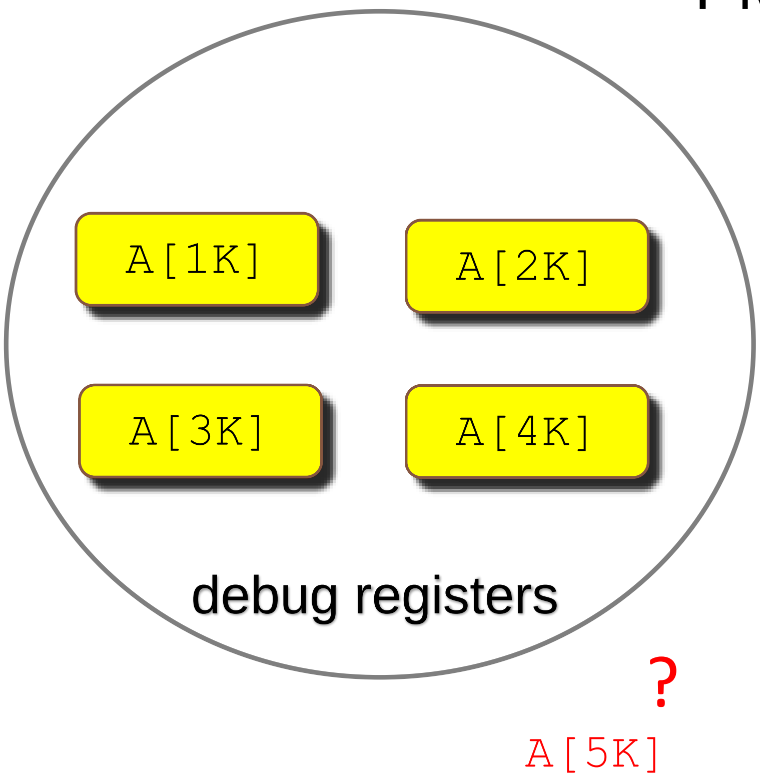
PMU samples every 1K memory stores




```
for (int i=1; i<=10K; i++){
  A[i] = 0;    i=4K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

PMU samples every 1K memory stores

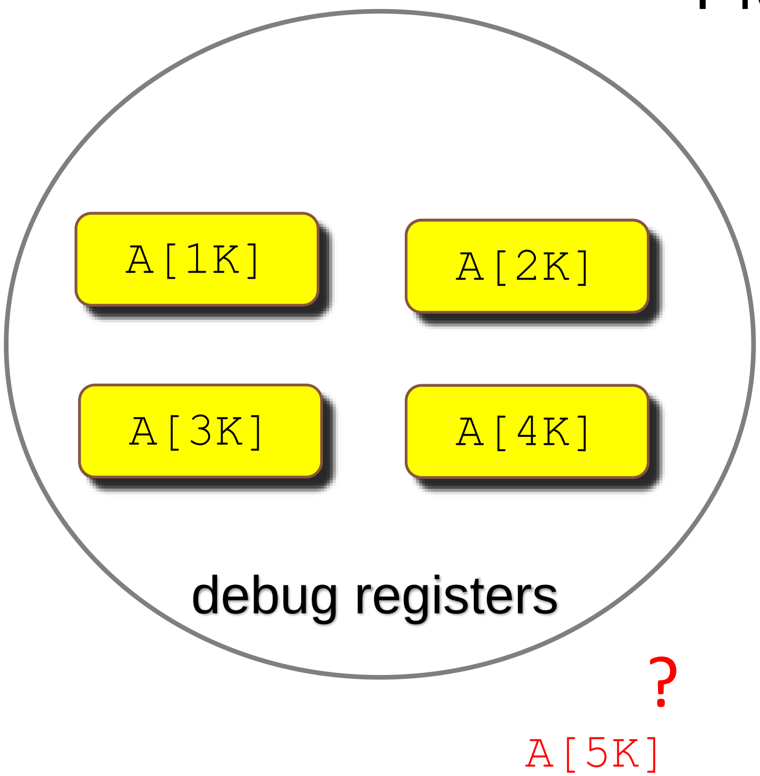



```
for (int i=1; i<=10K; i++){
  A[i] = 0;     i=5K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

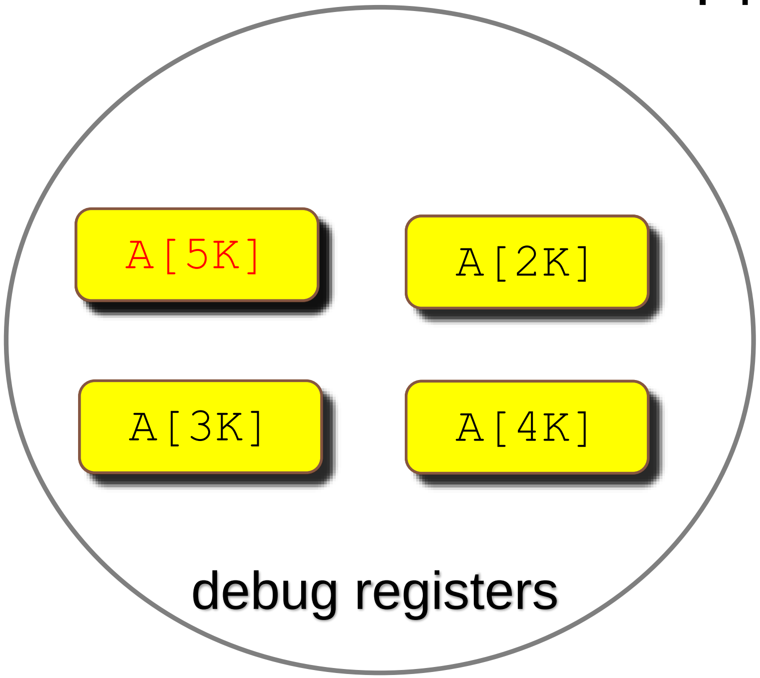



```
for (int i=1; i<=10K; i++){
   A[i] = 0;      i=5K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

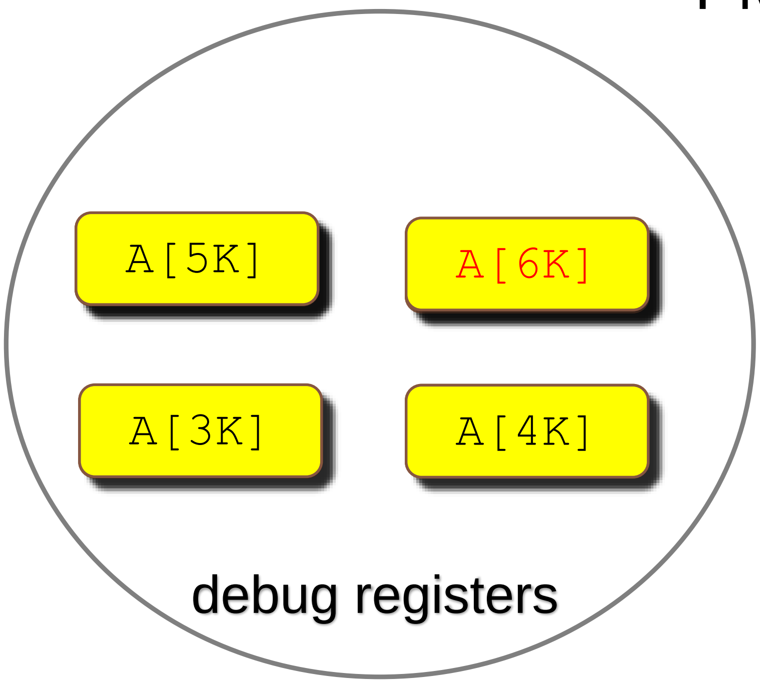



```
for (int i=1; i<=10K; i++){
   A[i] = 0;     i=5K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

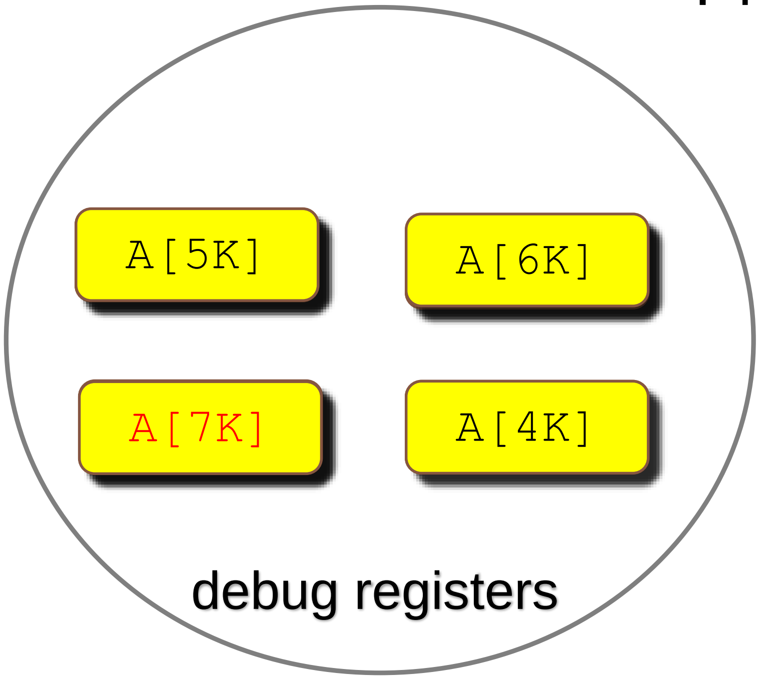



```
for (int i=1; i<=10K; i++){
  A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

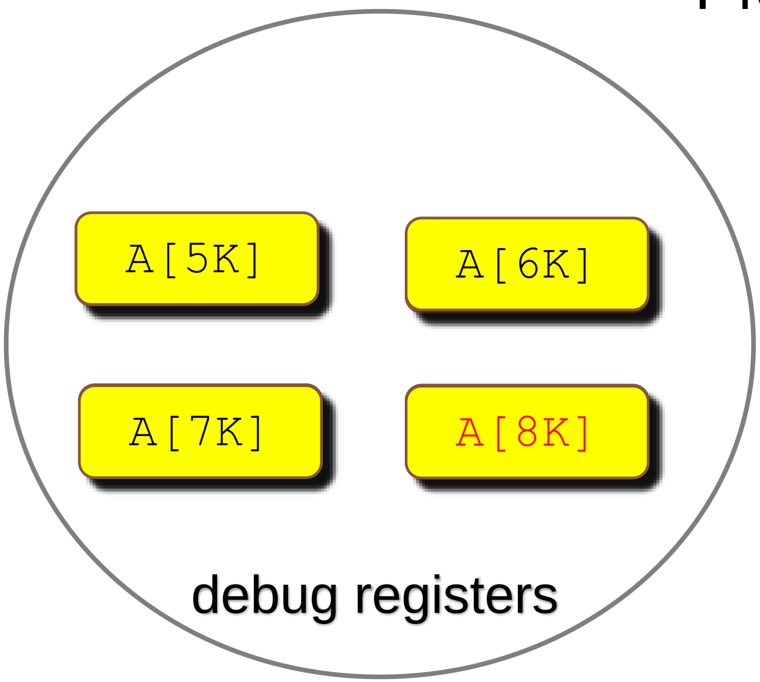



```
for (int i=1; i<=10K; i++){
   A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

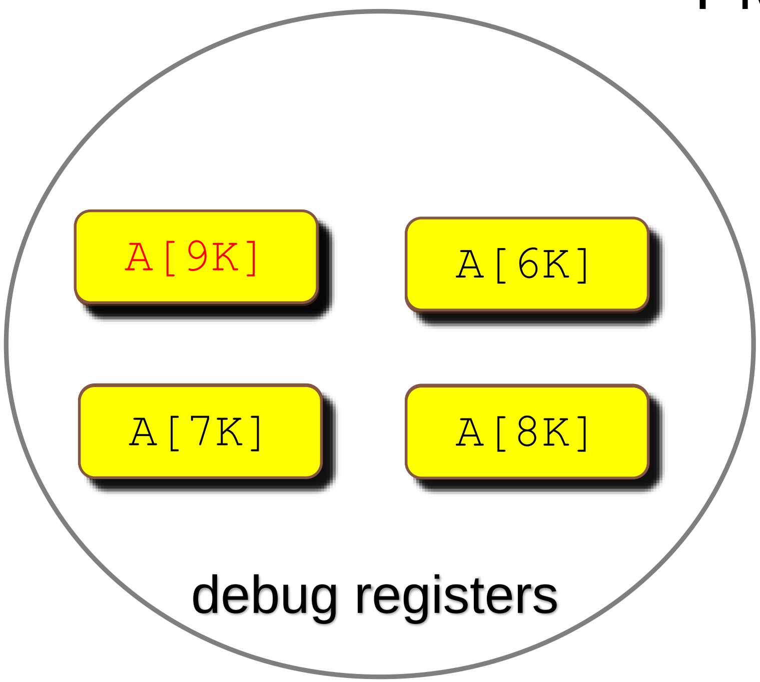



```
for (int i=1; i<=10K; i++){
  A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

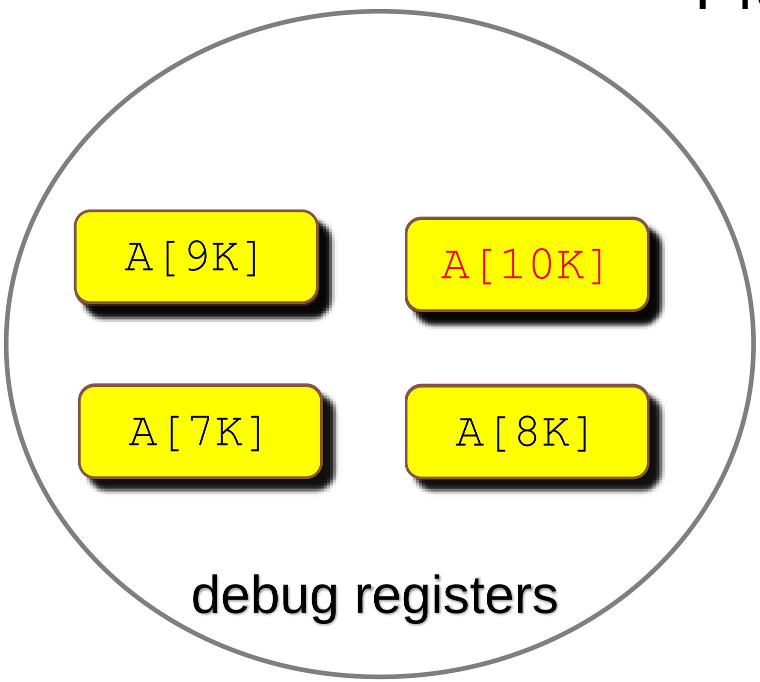



```
for (int i=1; i<=10K; i++){
  A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores

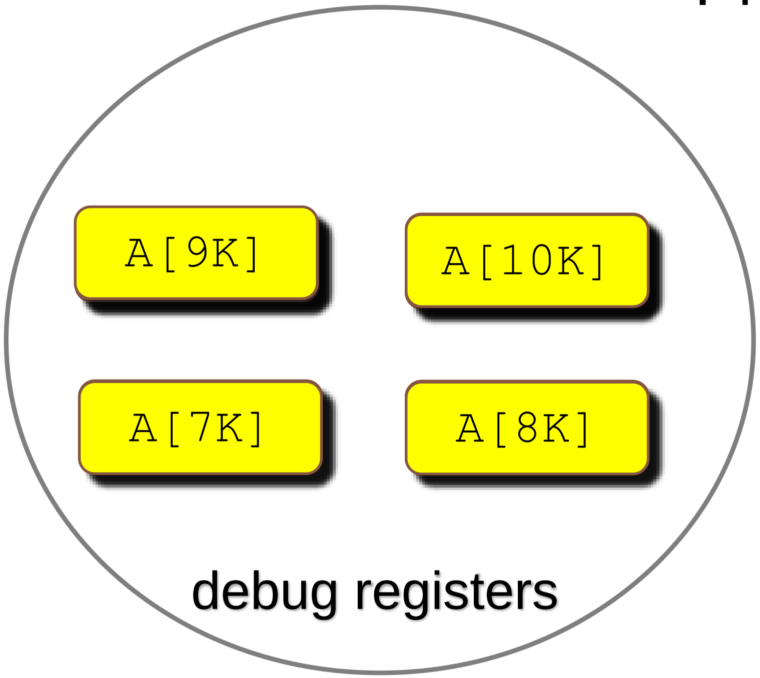



```
for (int i=1; i<=10K; i++){
  A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores



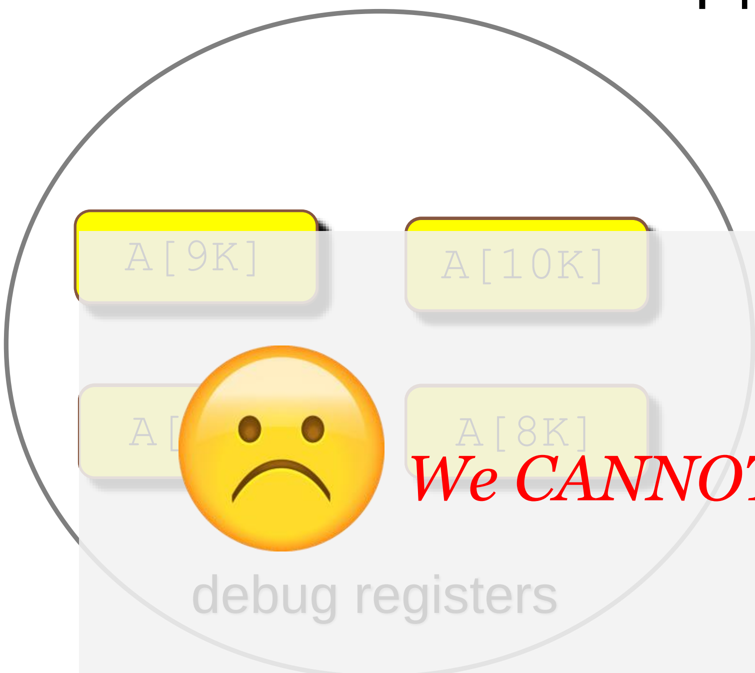

```
for (int i=1; i<=10K; i++){
   A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;      j=1K
}
```

Wait? We should have detected a reuse of `A[1K]` if it were not kicked out from debug registers.

Challenge

- Handle a limited number of debug registers
  - Strategy: replace the oldest one

PMU samples every 1K memory stores




```
for (int i=1; i<=10K; i++){
   A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;      j=1K
}
```

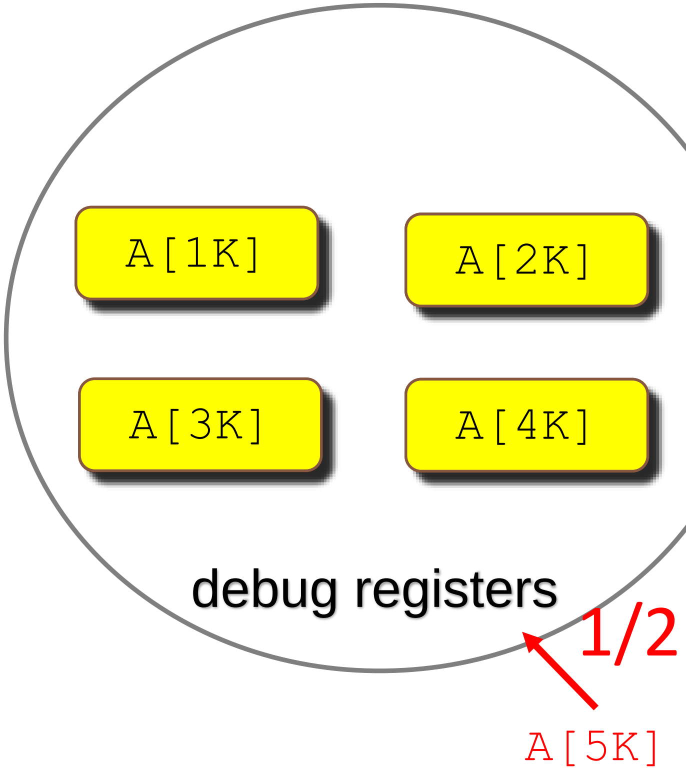
We CANNOT detect any reuse of `A`

Wait? We should have detected a reuse of `A[1K]` if it were not kicked out from debug registers.

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores

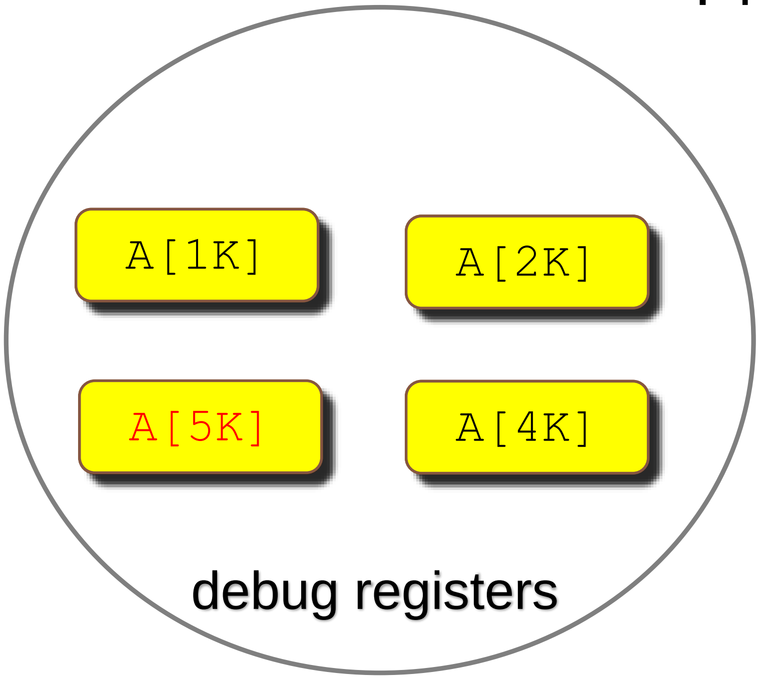



```
for (int i=1; i<=10K; i++){
  A[i] = 0;     i=5K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
  A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores

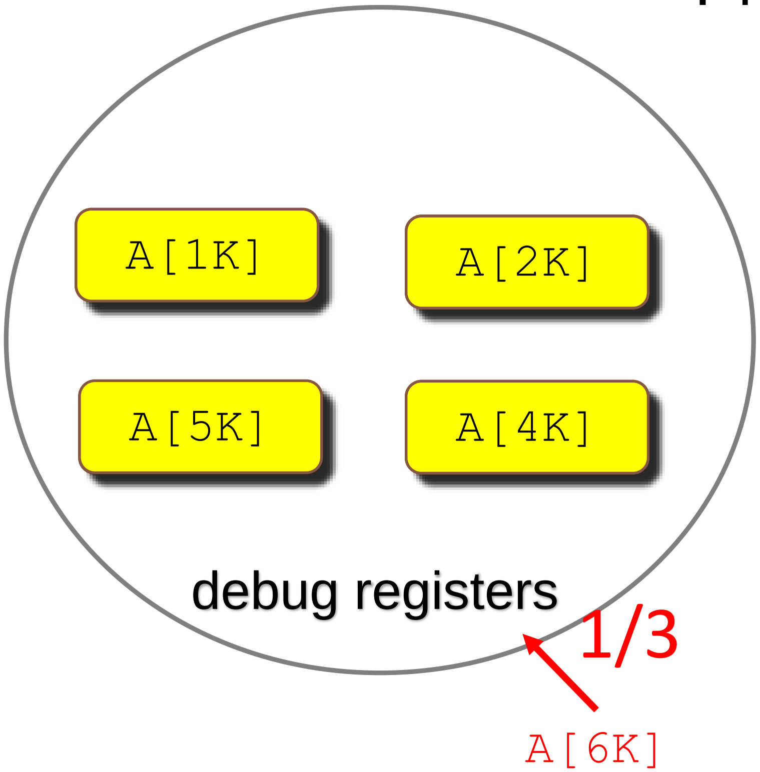



```
for (int i=1; i<=10K; i++){
   A[i] = 0;      i=5K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores

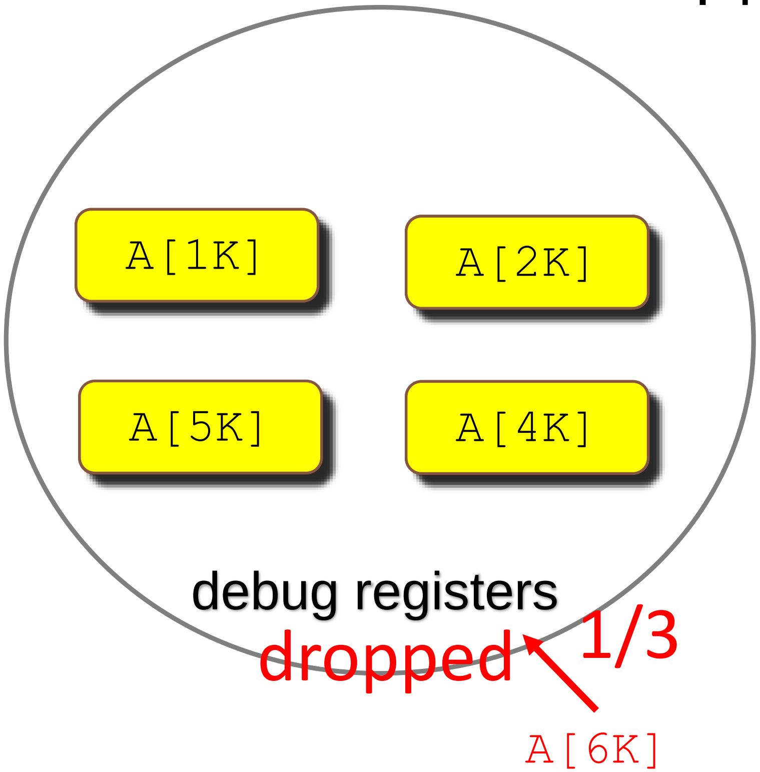



```
for (int i=1; i<=10K; i++){
   A[i] = 0;     i=6K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores

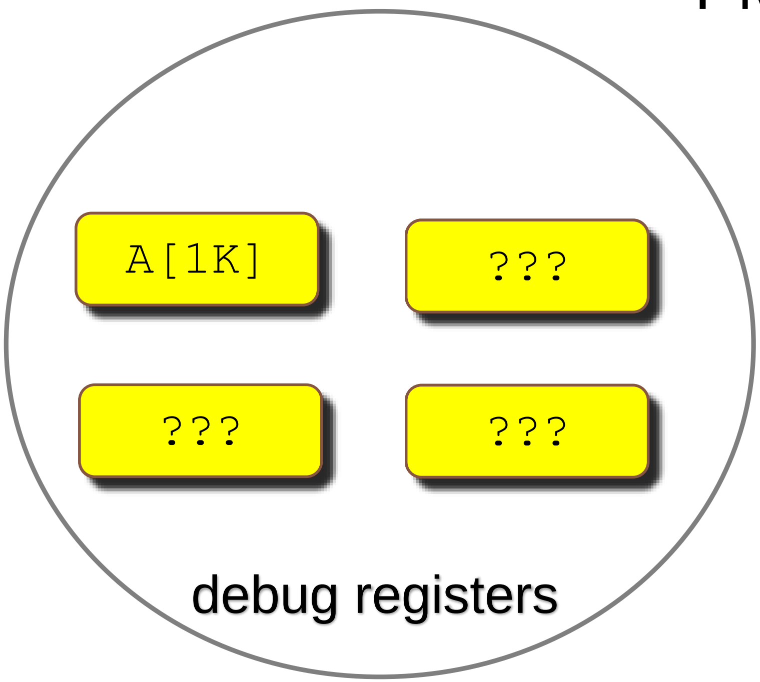



```
for (int i=1; i<=10K; i++){
   A[i] = 0;    i=6K
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores

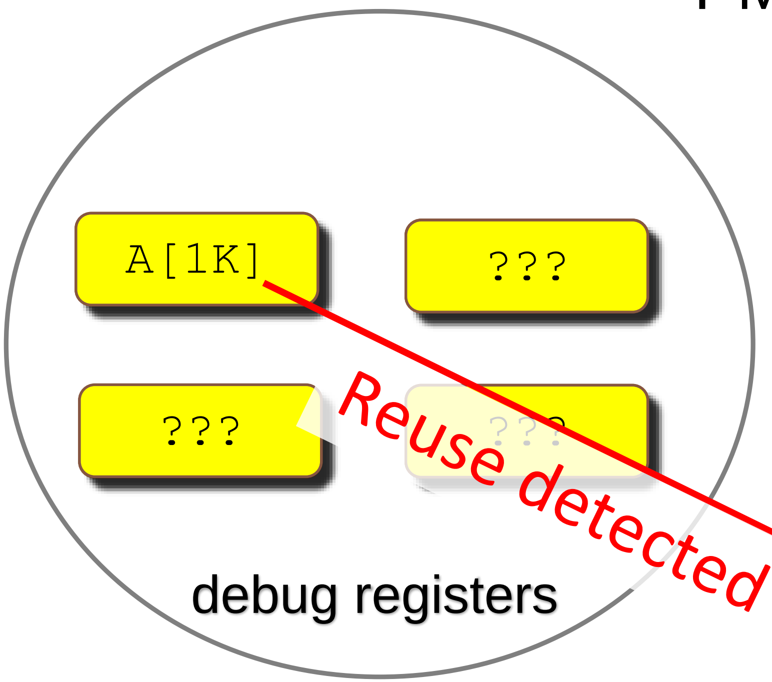



```
for (int i=1; i<=10K; i++){
    A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
    A[j] = 0;   j=1K
}
```

Challenge

- Handle a limited number of debug registers
  - Strategy: probabilistically get monitored

PMU samples every 1K memory stores



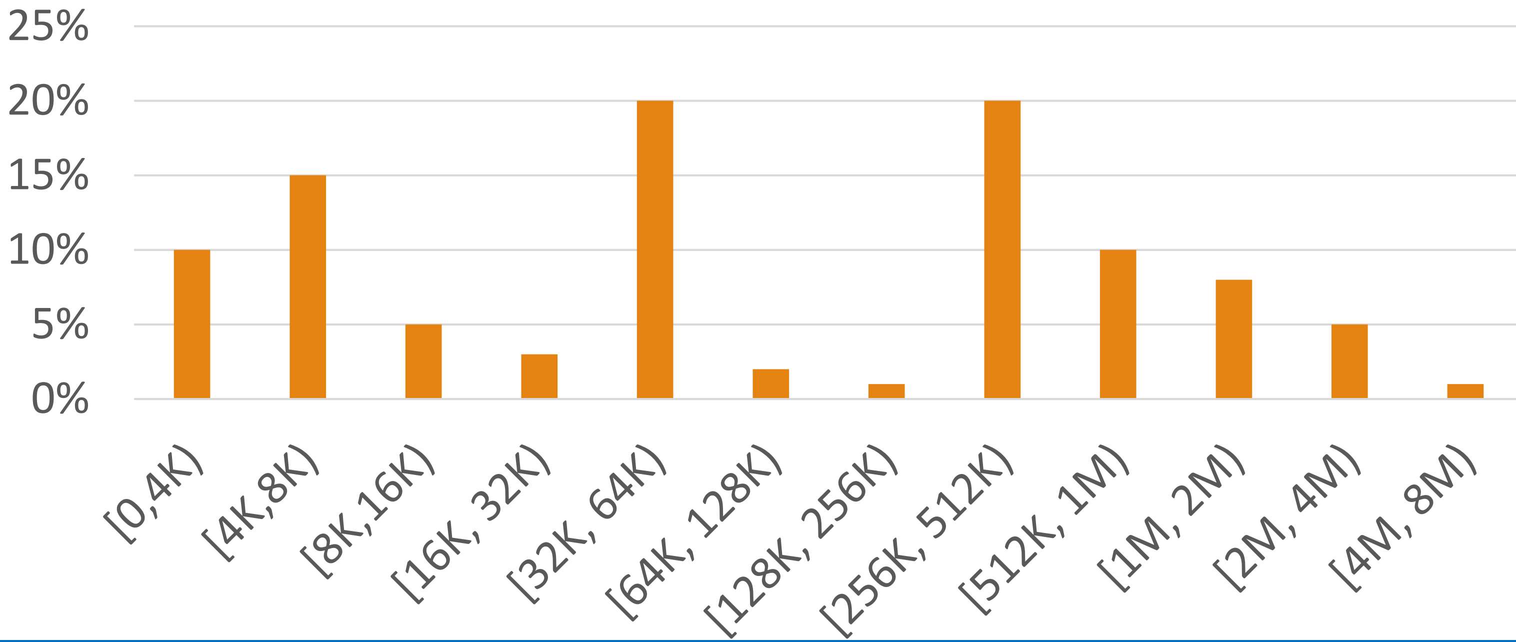

```
for (int i=1; i<=10K; i++){
   A[i] = 0;
}
// All elements of A are
// reused
for (int j=1; j<=10K; j++){
   A[j] = 0;   j=1K
}
```

Reservoir Sampling

If there is a free register, use it.
Otherwise, probabilistically replace one of monitored addresses

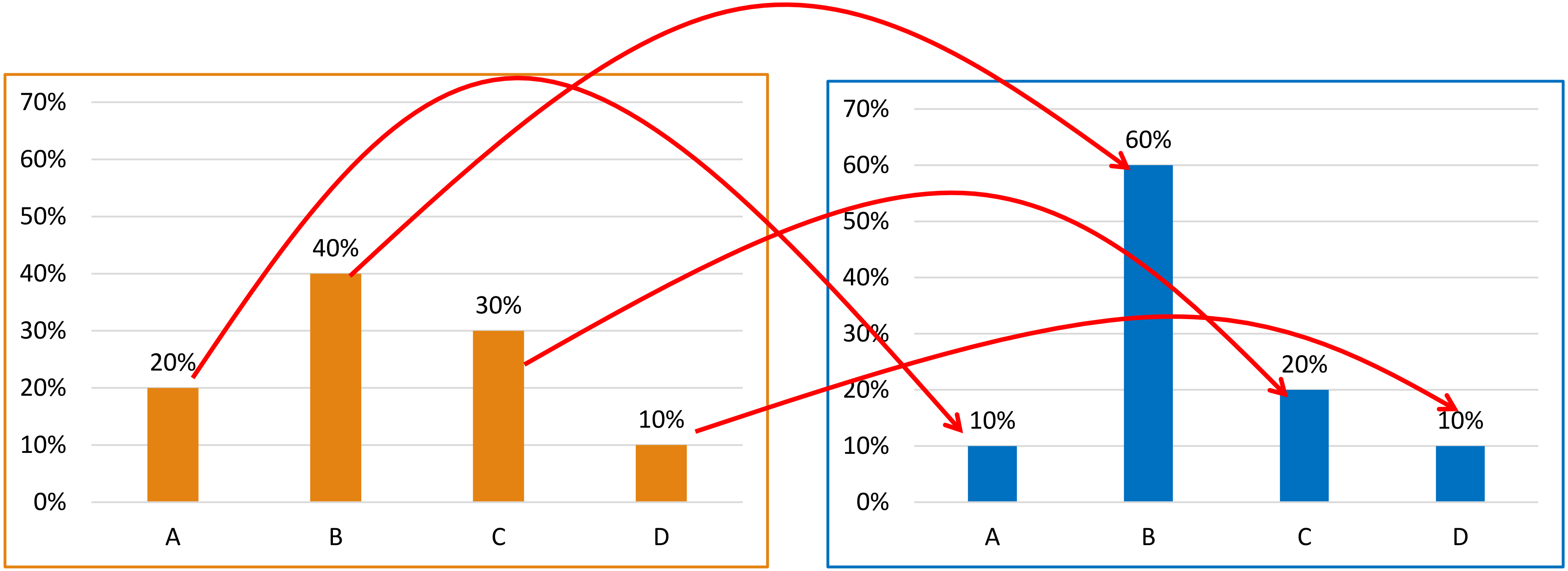
Evaluation on SPEC CPU2006

- Overhead
  - ~5%(time), ~5MB / thread (memory)
- Accuracy
  - Baseline: Intel PIN tool instruments every memory access
  - How similar a measured (estimated) histogram is to the baseline?



Evaluation on SPEC CPU2006

- Similarity
  - $S \in [0,1]$
  - $S = 1$, exactly the same



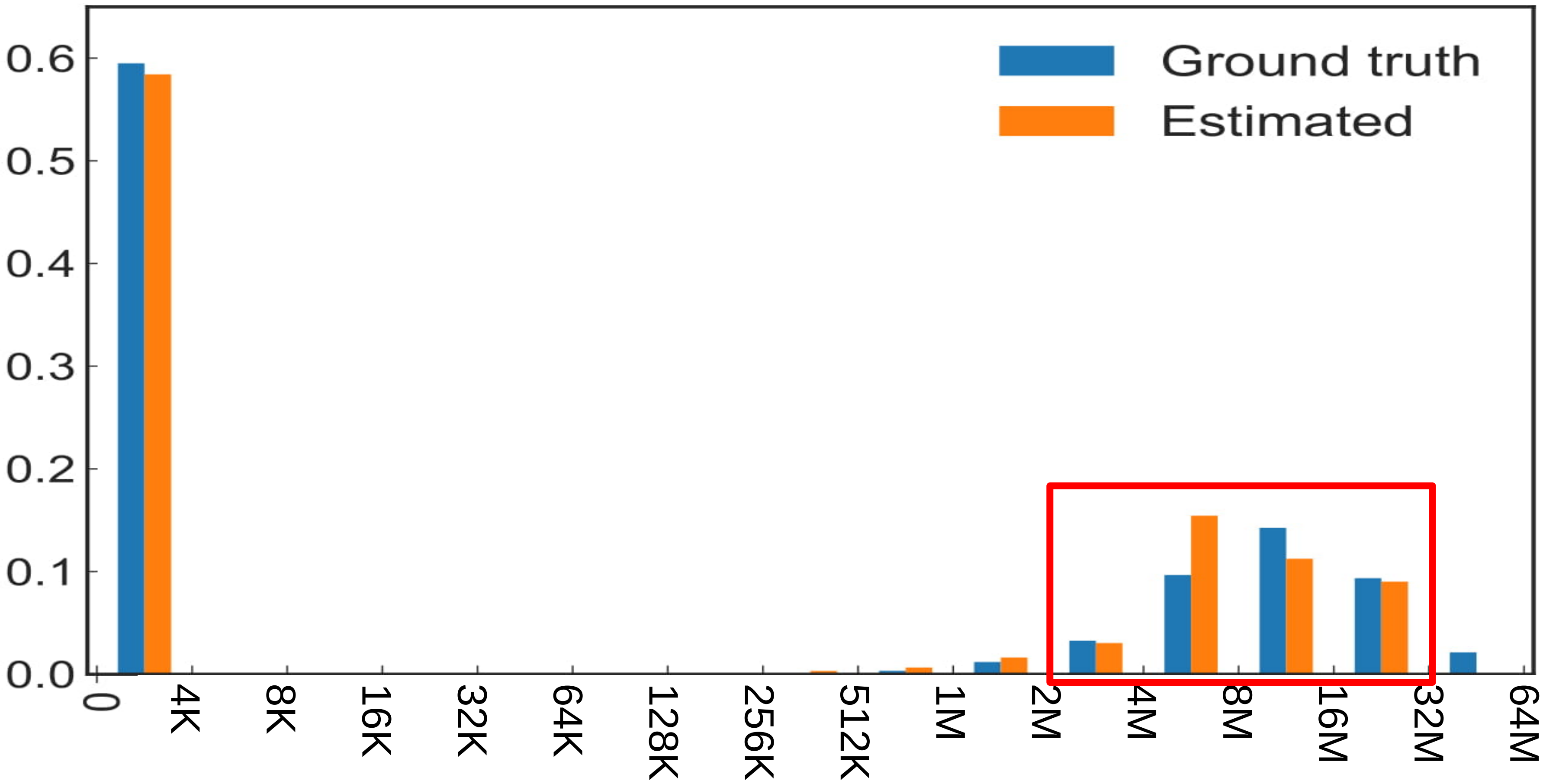
$$S = 1 - \frac{|0.2 - 0.1| + |0.4 - 0.6| + |0.3 - 0.2| + |0.1 - 0.1|}{2} = 0.8$$

Evaluation on SPEC CPU2006

- Time distance histogram accuracy
  - Median > 96%
- Stack distance histogram accuracy
  - Median > 90%
- Inaccuracy reason
  - Sparse reservoir sampling
  - Model problem
  - PMU imprecision

Evaluation on SPEC CPU2006

Estimated Stack Reuse Histogram



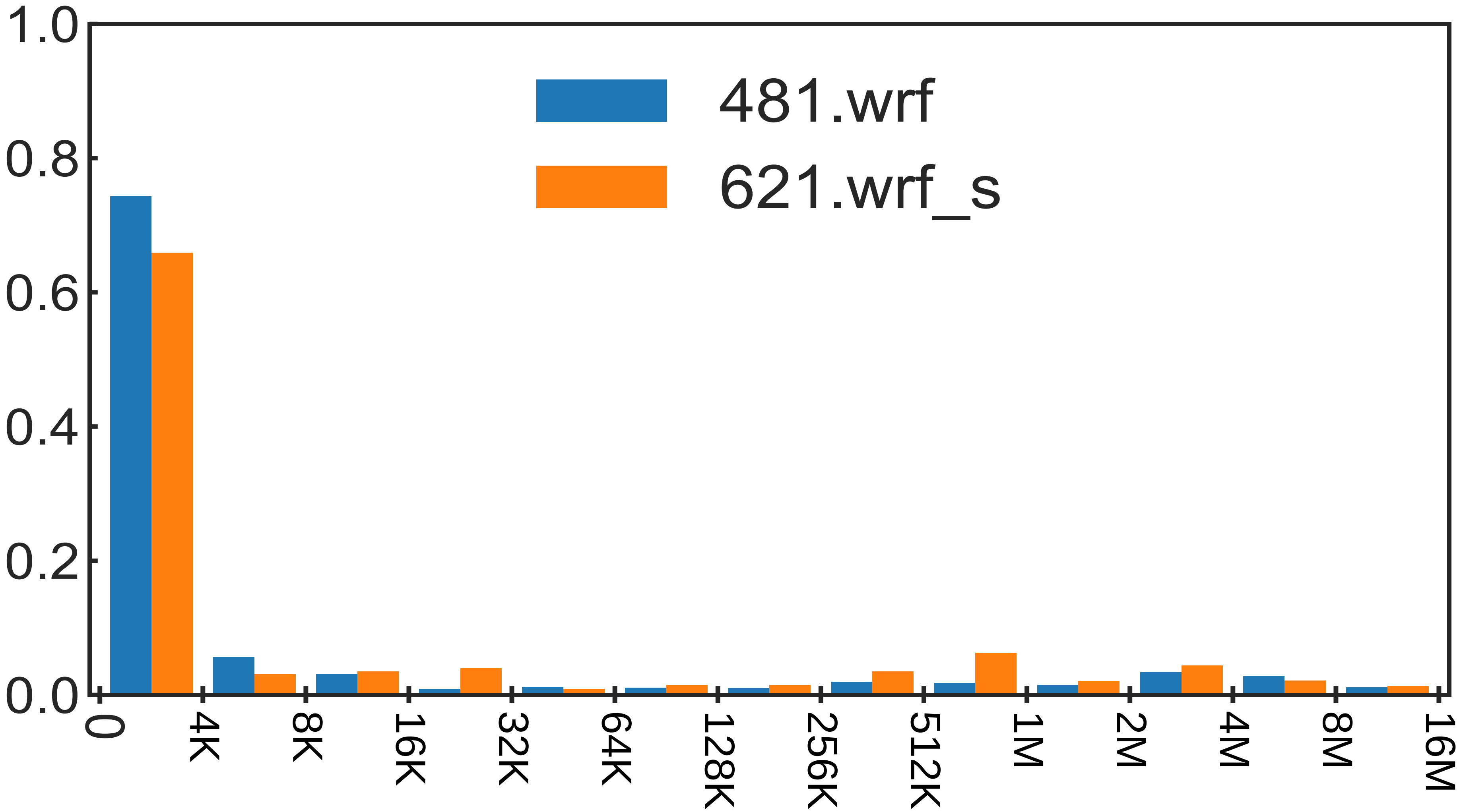
milc (SPEC CPU2006)

Evaluation on SPEC CPU2017

- First to study data locality of SPEC CPU2017
- Plot stack reuse histograms of all individual benchmarks
- SPEC CPU2006 vs. 2017
  - SPEC CPU2006 (4xx series)
  - SPEC CPU2017 speed (6xx series)

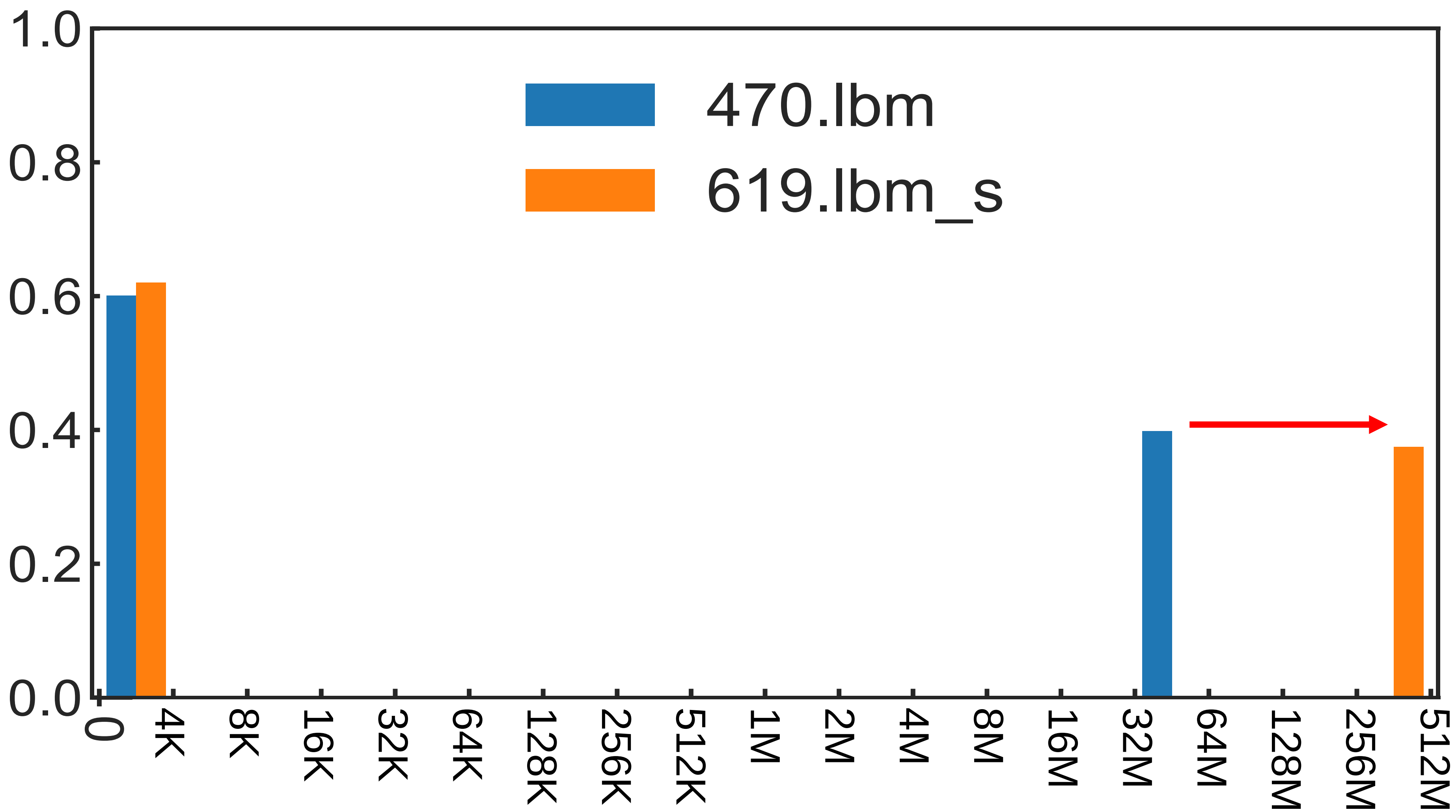
SPEC CPU2006 → 2017

- Unchanged



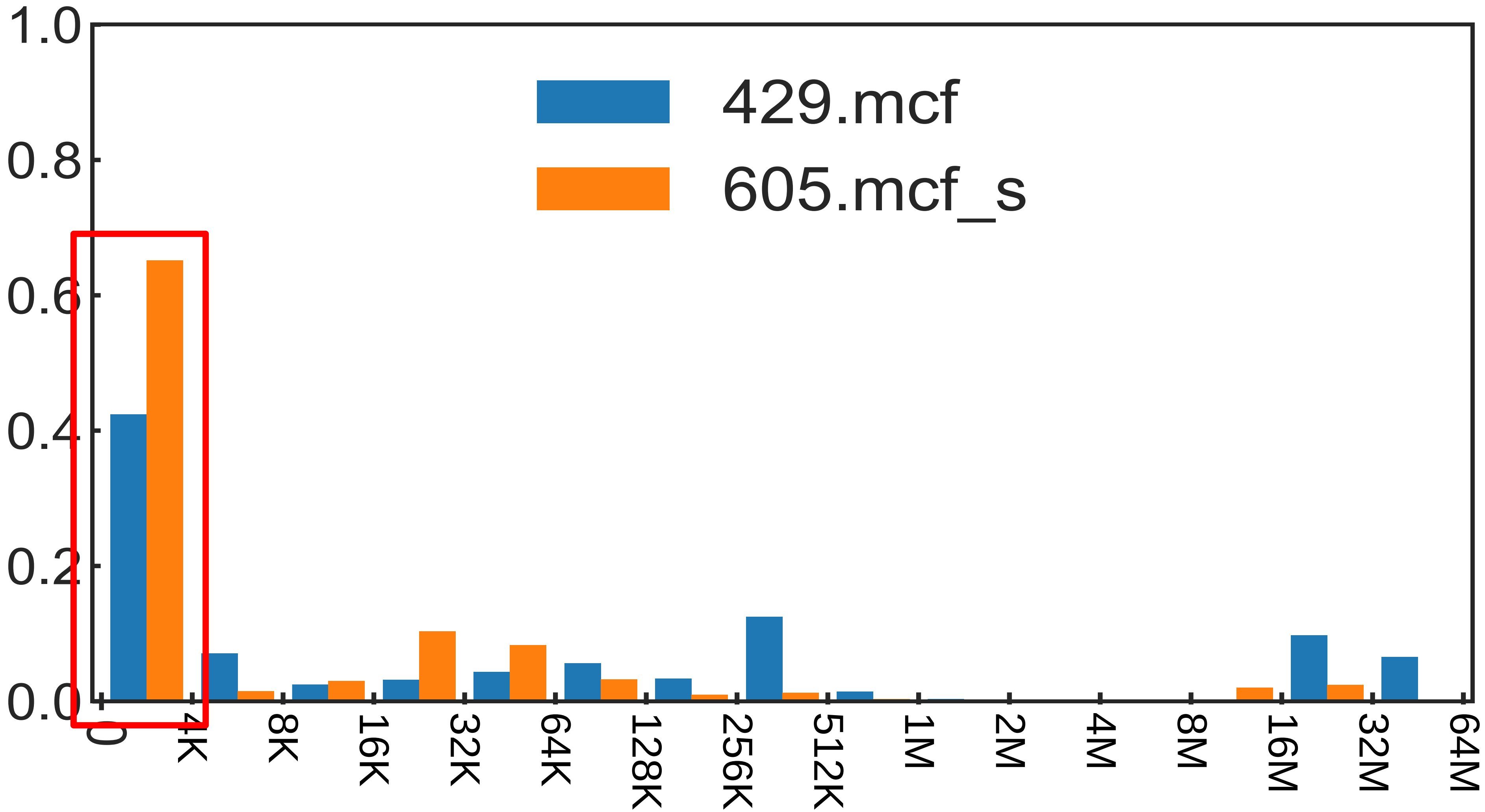
SPEC CPU2006 → 2017

- Reuse distance has increased dramatically



SPEC CPU2006 → 2017

- Reuse distance has decreased



Code Optimization

- Strategy
  - Pinpoint high-penalty cache misses
  - Analyze with reuse distance
- Speedup overview

Programs	Improved locality	Optimization	Speed-up
lulesh	temporal	fuse loops	1.54X
botsspar	spatial & temporal	interchange loop iterations within a nested loop	3.45X
backprop	Spatial	interchange loop iterations within a nested loop	1.52X
srad_v1	Spatial	interchange loop iterations within a nested loop	1.80X
sweep3d	spatial	transpose arrays	1.04X

Conclusions

- RDX
  - Lightweight, sampling-based
  - Measures time & stack distance of the whole program
  - Guides optimization related to locality and cache performance
  - Relies on hardware performance units and hardware debug registers
- Characterization
  - SPEC CPU2006
  - SPEC CPU2017
- Optimization

Questions?