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 $\leq 2$, 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

- Use Performance Monitor Units (PMU) to sample LOAD and STORE instructions
- Record effective address of each access
- Use debug registers to detect the reuse position of a memory location
- Measure time distance of the sampled address
- Each data location is accessed independently
- Statistically estimate reuse distance histogram from time distance

Sample memory access address

Measure time distance of the sampled address

Time distance → reuse distance
Sample memory access address

Measure time distance of the sampled address

Time distance $\rightarrow$ reuse distance
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)
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.

PMU sample

memory access

Seq No.  1  2  ...  99  100  101  ...  199  200  201  ...  232  233  234  ...

debug registers

a,100

... ? a ? ... ? ? ? ... ? ? ? ...
RDX – Measure Time Distance

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

PMU sample

Seq No. 1 2 ... 99 100 101 ... 199 200 201 ... 232 233 234 ...

memory access

debg registers

a, 100 e, 200

... ? a ? ... ? e ? ... ? ? ...
RDX – Measure Time Distance

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

```
Seq No.  1   2   ...   99  100  101   ...   199  200  201   ...   232  233  234   ...
```

```
...  ?  a  ?  ...  ?  e  ?  ...  ?  a  ?  ...
```

Reuse found!

dep registers

```
a,100  e,200
```
Use debug register to measure time distance
PMU samples every 100 memory references.

Time distance of \( a \) is 233-100=133
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

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

- 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 – Time → Reuse

- How is time distance related to stack distance?

<table>
<thead>
<tr>
<th>Time distance</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
How is time distance related to stack distance?

<table>
<thead>
<tr>
<th>Time distance</th>
<th>Occurrence</th>
<th>Reuse distance</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
RDX – Time $\rightarrow$ Reuse

- How is time distance related to stack distance?

![Diagram showing memory access with time and stack distances]

Not feasible to enumerate all the possibilities for real programs
RDX – Time $\rightarrow$ Reuse

- Statistically convert time distance to reuse distance

<table>
<thead>
<tr>
<th>Locality Approximation Using Time (POPL’07)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Assumption</strong></td>
</tr>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
</tbody>
</table>
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 $\rightarrow$ reuse distance
- Each data location is accessed independently
- Statistically estimate reuse distance histogram from time distance
PMU samples every 1K memory stores

```c
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;
}
```
PMU samples every 1K memory stores

```c
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;
}
```
PMU samples every 1K memory stores

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

debug registers

PMU samples every 1K memory stores

```c
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

```plaintext
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;
}
```


depth registers

?
Challenge

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

PMU samples every 1K memory stores

```c
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;
}
```

debug registers
Challenge

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

PMU samples every 1K memory stores

```c
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

```c
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

```c
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;
}
```

PMU samples every 1K memory stores
Challenge

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

PMU samples every 1K memory stores

```c
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

```c
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

```c
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

```c
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

```c
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;  // we should have detected a reuse of A[1K]
```

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

**We CANNOT detect any reuse of A**
Challenge

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

PMU samples every 1K memory stores

```java
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: probabilistically get monitored

PMU samples every 1K memory stores

```c
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

```c
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

```c
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;
}
```


debug registers dropped \( \frac{1}{3} \)

A[6K]
Challenge

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

PMU samples every 1K memory stores

```cpp
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

```cpp
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;
```

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

Reservoir Sampling

debug registers

![debug registers diagram]
Evaluation on SPEC CPU2006

- **Overhead**
  - \(~5\%\) (time), \(~5\) MB / 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

- 481.wrf
- 621.wrf_s
Reuse distance has increased dramatically

SPEC CPU2006 → 2017

470.lbm
619.lbm_s
SPEC CPU2006 → 2017

- Reuse distance has decreased
Code Optimization

- **Strategy**
  - Pinpoint high-penalty cache misses
  - Analyze with reuse distance

- **Speedup overview**

<table>
<thead>
<tr>
<th>Programs</th>
<th>Improved locality</th>
<th>Optimization</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>lulesh</td>
<td>temporal</td>
<td>fuse loops</td>
<td>1.54X</td>
</tr>
<tr>
<td>botsspar</td>
<td>spatial &amp; temporal</td>
<td>interchange loop iterations within a nested loop</td>
<td>3.45X</td>
</tr>
<tr>
<td>backprop</td>
<td>Spatial</td>
<td>interchange loop iterations within a nested loop</td>
<td>1.52X</td>
</tr>
<tr>
<td>srad_v1</td>
<td>Spatial</td>
<td>interchange loop iterations within a nested loop</td>
<td>1.80X</td>
</tr>
<tr>
<td>sweep3d</td>
<td>spatial</td>
<td>transpose arrays</td>
<td>1.04X</td>
</tr>
</tbody>
</table>
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?