Low Overhead Yet Accurate Profiling Tools for Multicore

Dr. Didem Unat
Koç University, Istanbul, Turkey
27 Jan 2022
Modern HPC Applications

- Employ multi-socket, multicore, many-core CPUs within a node
- Use MPI+Threads
  - MPI for communication among nodes
  - OpenMP or other threading models for intra-node communication
- Do explicit inter-process communication
  - Managed via message passing (e.g., MPI) Send/Recv primitives
- Do implicit inter-thread communication
  - Hidden by standard load/store CPU instructions

Regardless of communication type, data transfer is dominant in performance and energy consumption.

Thus, we need to tools to monitor data locality for multicore!
Precise-Event Sampling in PMUs

• **Hardware feature** in commodity CPUs
  - Extends Performance Monitoring Units (PMUs)
  - PMUs consist hardware counters + model-specific registers (MSRs)

• **Sampling hardware or software events** periodically based on user-defined sampling period

• **Attributing** those samples **accurately** to the **instructions** that trigger them.

• Examples: Intel PEBS, AMD IBS, PowerPC MRK, ARM SPE
Profiling Tools based on Event Sampling


- **ReuseTracker**: “ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer”, ACM TACO and HiPEAC Conference, June 2022

- **AMD vs Intel**: M. A. Sasongko, M. Chabbi, P. H. J. Kelly, D. Unat, in preparation for submission
Need Communication Detection Tools

MPI communication matrix for LULESH via EZTrace

Inter-thread communication matrix

?
Need Communication Detection Tools

MPI communication matrix for LULESH via EZTrace

Our Contribution: ComDetective

- Nearest neighbors
- Distant neighbors
- All-to-one (master) communication
Why Detect Inter-Thread Communication?

- Identify possible sources of performance bottlenecks
- Help explain why one threading library is better than another
  - e.g. Intel OpenMP vs GNU OpenMP
- Guide performance optimizations such as
  - thread binding
  - data structure modification
  - false sharing elimination
- Hardware design: on-chip network design, cache coherence protocol
Challenges

- Inter-process communication detection in MPI is relatively straightforward

  ```c
  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  ```
  
  ```c
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  ```
  
  There is **4 bytes data transfer** from process 0 to process 1

- Exact inter-thread communication detection poses some challenges
  - requires **interception of load and store** operations
  - incurs **huge space and time overheads** if all load and store operations are intercepted
  - **dilates execution** and changes program behavior
  - **scales poorly** with increasing number of threads
ComDetective

**Accurate**
Validated against several benchmarks and HPC applications

**Lightweight**
Space overhead (1.27x) and time (1.3x) overhead

**Sampling based**
Uses ready-available hardware events in commodity CPUs

**Differentiates kind of communication**
True sharing (necessary) vs. false sharing (unnecessary)

**Data objects**
Attributes communication to program data objects

**Open source**
https://github.com/ParCoreLab/ParCoreTools
Inter-Thread Communication

- Occurs in multi-threaded programs or hybrid programs (e.g. MPI+OpenMP hybrid)
- Occurs at CPU cache line granularities
Inter-Thread Communication

- Memory access by CPU 0
Inter-Thread Communication

- Memory access by CPU 1
• This type of communication is called **true sharing**
Another possible type is **false sharing**

Threads 0 and 1 access different memory regions in the same cache line.
In addition to communication matrix, ComDetective also produces true sharing and false sharing matrices.

It took only $1.28x$ performance and $1.11x$ memory footprint overhead to generate these matrices with ComDetective.
Problem with existing tools

1. **Not accurate:**
   - Distortion of parallel schedule among threads by binary instrumentation (Pericas, et al. ICS 2014)
   - Unable to capture actual communication pattern in real hardware
Problems with existing tools

2. Huge time and memory overheads:
   - Compiler-assisted instrumentation (Mazaheri, et al. ICPP 2015, Mazaheri, et al. ICPP 2018)

3. Intrusive and detecting communication at memory page level:
Design Components: PMU

**PMU**: Special registers that count low-level events, such as loads or stores.

**Sampling**: PMUs can be configured to trigger interrupt for every N events.
Debug Registers: Filled with a memory address and length.
Sends a trap when the memory region specified by the address and length is accessed.
Design Components: perf_event

perf_event: Allows user applications to **configure and access PMUs** and **debug registers**
Design Components: perf_event

perf_event: Allows user applications to **configure and access PMUs** and **debug registers**
Design Components: ComDetective

A function that runs when **sampling** happens
Design Components: ComDetective

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bulletin Board:** A hash table that publishes some of the sampled data

**perf_event**  **sample_handler**  **sample_handler**  **perf_event**

**PMUs**

- core 0

**PMUs**

- core 1

**Debug registers**
Design Components: ComDetective

A function that runs when debug register trap happens

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

perf_event   sample_handler   trap_handler   trap_handler   sample_handler   perf_event

PMUs

core 0

Debug registers

PMUs

core 1

Debug registers
Design Components

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread T0
- perf_event
- sample_handler
- trap_handler

Thread T1
- trap_handler
- sample_handler
- perf_event

PMUs
- core 0
- core 1

Debug registers
perf_event is configured so that loads and stores to be sampled for every N events.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

An Example Workflow

Thread T0

- perf_event
- sample_handler
- trap_handler

PMUs

- core 0

Debug registers

Thread T1

- trap_handler
- sample_handler
- perf_event

PMUs

- core 1

Debug registers
**Workflow**

*Interrupt* happens in core 0 after N events.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Thread T0

- perf_event
- sample_handler
- trap_handler

PMUs

- core 0

Debug registers

Thread T1

- trap_handler
- sample_handler
- perf_event

PMUs

- core 1

Debug registers
If it is a store event, published it on bulletin board.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>M0, L0, timestamp, T0</td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Workflow

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>M0, L0, timestamp, T0</td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

*Interrupt* happens in core 1. The triggering event is a *store to memory address M1.*
Workflow

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>M0, L0, timestamp, T0</td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Check whether cache lines match!

Thread T0
- perf_event
- sample_handler
- trap_handler

Thread T1
- trap_handler
- sample_handler
- perf_event

PMUs
- core 0

PMUs
- core 1

Debug registers
Check hash table entry whether there is the entry is ‘recent’.

If C0’s entry is ‘recent’, communication is detected between T0 and T1.
If C0’s entry is not ‘recent’, replace the entry with M1.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>M1, L1, timestamp, T1</td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Thread T0
- perf_event
- sample_handler
- trap_handler

Thread T1
- trap_handler
- sample_handler
- perf_event

PMUs
- core 0
- Debug registers

PMUs
- core 1
- Debug registers
Another store sample happens on address M3.

No matching cache line and all entries are ‘recent’, so none can be replaced.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>.....</td>
</tr>
<tr>
<td>...</td>
<td>.....</td>
</tr>
<tr>
<td>...</td>
<td>.....</td>
</tr>
</tbody>
</table>
Workflow

Set up debug register from a randomly selected entry in the hash.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread T0
- perf_event
- sample_handler
- trap_handler

Thread T1
- trap_handler
- sample_handler
- perf_event

PMUs
- core 0
- Debug registers

PMUs
- core 1
- Debug registers
Workflow

When trap in a debug register happens, communication matrices are updated.

<table>
<thead>
<tr>
<th>key</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread T0
- perf_event
- sample_handler
- trap_handler

Thread T1
- trap_handler
- sample_handler
- perf_event

PMUs
- core 0

PMUs
- core 1

Debug registers

Debug registers
Snapshot of PARSEC Matrices (only 6 shown)

(a) Blackscholes  
(b) Bodytrack  
(c) Canneal  
(d) Dedup  
(e) Facesim  
(f) Ferret
Blacksholes, financial analysis benchmark

Splits the price options among threads where each thread can process the options independently from each other
CORAL Benchmarks

- MPI
- AMG
- MiniFE
- PENNANT
- QuickSilver
- VPIC
Use Cases: Code Refactoring

- **False sharing** in `streamcluster` happens on `pthread_mutex_t` typed variables
  - 6% improvement is achieved after we put paddings among attributes in `pthread_mutex_t` struct
- **False sharing** in `fluidanimate` happens on a `pthread_cond_t` typed variable
  - 13% improvement is achieved after we put paddings among attributes in `pthread_cond_t` struct
Reuse and Invalidation in Private Cache

- **Reuse:**
  - access sequence: a, b, c, b, d, d, a
  - time distance of a: 5
  - reuse distance of a: 3

- **Invalidation:**
  - T1: a, b, c, b, d, d, a
  - T1 and T2 are in different cores.
  - T2: p, q, r, a, ...
  - (store)
Reuse and Invalidation in Shared Cache

● **Reuse:**
  ○ T1: a, b, c, b, ........
  ○ T2: p, q, r, a, ......  
    (load/store)
  
  T1 and T2 are in different cores but still in the same socket.

● **Invalidation:**
  ○ T1: a, b, c, b,......
  ○ T3: p, q, r, a, ......  
    (store)
  ○ T3: p, q, r, a, ......  
    (store)
  
  T1 and T2 are in different cores and in different sockets.
## Comparison with Existing Tools

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Overhead</td>
<td>High (&gt;100x)</td>
<td>30x</td>
<td>&lt;1.4x</td>
<td>49x</td>
<td>2.03x</td>
<td>2.9x</td>
</tr>
<tr>
<td>Space Overhead</td>
<td>Not reported</td>
<td>Not reported</td>
<td>Not reported</td>
<td>40x</td>
<td>2.8x</td>
<td>2.8x</td>
</tr>
<tr>
<td>Intra-thread Profiling Accuracy</td>
<td>Close to 100%</td>
<td>96%</td>
<td>High but not quantified</td>
<td>Close to 100%</td>
<td>90%</td>
<td>92%</td>
</tr>
<tr>
<td>Primary Method</td>
<td>Cycle accurate simulator</td>
<td>Binary instrumentation</td>
<td>PMU</td>
<td>Binary instrumentation</td>
<td>PMU</td>
<td>PMU</td>
</tr>
<tr>
<td>Profiling Private Cache</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Profiling Shared Cache</td>
<td>YES</td>
<td>YES</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>Cache Line Invalidation</td>
<td>YES</td>
<td>YES</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>Open Source</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
ReuseTracker

Fast

Accurate

Sampling based

Multithreads

introducing low time overhead

Verified against a set of microbenchmarks with known ground truths

Uses ready-available hardware events in commodity CPUs

Profiles reuse distance in private and shared caches by also detecting cache line invalidations

Code line attribution
Attributes uses and reuses to source code lines

Open source
https://github.com/ParCoreLab/ParCoreTools
Accuracy Verification and Overhead

- **Accurate**: 92% on average using synthetic benchmarks with configurable patterns.
- **Fast**: 2.9x runtime and 2.8x memory overheads (from 10 PARSEC benchmarks)
Summary

**ComDetective:** Inter-thread communication analyzer, Published at SC19.

**ReuseTracker:** Reuse distance analyzer for multithreaded code, Published at ACM TACO and will be presented in HiPEAC 2022.

Publicly available repository of the profiling tools and benchmarks:

- [https://github.com/ParCoreLab/ParCoreTools](https://github.com/ParCoreLab/ParCoreTools)

Acknowledgement: The work is supported by the Scientific and Technological Research Council of Turkey (TUBITAK), Grant no. 120E492. Dr. Didem Unat is supported by the Royal Society-Newton Advanced Fellowship.