ComDetective: A Lightweight Tool for Detecting Inter-Thread Communication

Muhammad Aditya Sasonko (Koç University), Milind Chabbi (Scalable Machines) Palwisha Akhtar (Koç University), Didem Unat (Koç University)

parcorelab.com
About me?
Research Interests

- Address software challenges of emerging architectures
- Develop tools in collaboration with computational scientists

  - Programming models and runtime systems
    - Data locality is at the center
    - Focus on homogeneous and heterogeneous large-scale systems
    - Embrace asynchrony to scale on thousands of processors
  
  - Tools for performance monitoring and modeling
    - Design and develop tools for performance modeling and optimization on multicore and heterogeneous architectures

TiDA  Perilla  ExaSAT  TaskSanitizer  ComDetective
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.
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

- All-to-one (master) communication
- nearest neighbors
- distant neighbors
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
  - Process 0
    - `MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);`
  - Process 1
    - `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: Salient Features

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

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

- **Sampling-based**
  - Uses hardware performance monitoring units

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

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

- Open source: [https://github.com/comdetective-tools](https://github.com/comdetective-tools)
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
Inter-Thread Communication: Necessary

- This type of communication is called **true sharing**

![Diagram showing inter-thread communication and cache line sharing between Thread 0 and Thread 1 on CPUs 0 and 1.](image)
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 $1.28\times$ performance and $1.11\times$ memory footprint overhead to generate these matrices with ComDetective.
Existing Tools

- Prior works on identifying inter-thread communication employed **hardware simulators** or **binary instrumentation**
  - Suffered from **inaccuracy** or **high overhead**
  - Incurring **huge memory footprints** and **very slow**
  - Requires offline profiling
  - Not running on real hardware, so execution behavior can change.
  - Can be **intrusive** as it requires modification of kernel source code
Existing Tools

  - Can suffer from **large slowdown and memory overhead**
  - We have found that Numalize is **not accurate**.
Existing Tools

  - Can suffer from large slowdown and memory overhead
  - We have found that Numalize is not accurate.
We develop a tool to detect inter-thread communication called ComDetective. ComDetective is:

- **Fast** -- uses available hardware features; PMUs and debug registers
- **Accurate** -- has been validated in terms of correctness of total communication volume and correctness of point-to-point communication ratio

ComDetective also:

- Differentiates true sharing and false sharing communications -- by detecting if memory regions accessed by communicating threads overlap or not
- Associates communication matrices not only to the whole program but to program objects -- for global, stack, and heap objects
Outline

- Background Information on Inter-Thread Communication
- Motivation for Detecting Inter-Thread Communication
- Prior Arts
- Introduction to ComDetective
- Design Components
- Workflow
- Detailed Evaluation
Inter-thread communication occurs between two threads if,

Two threads access an address residing on the same cache line in a short interval

Question: How to detect cache line communication?
Inter-thread communication occurs between two threads if,

Two threads access an address residing on the same cache line in a short interval

Question: How to detect cache line communication?
- A thread can **sample** its memory accesses via hardware performance counters (address sampling)
- No load/store instrumentation $\Rightarrow$ super low overhead
• Question: how can another thread know if it is accessing the same address without instrumenting its loads and stores?
Big Picture

• Question: how can another thread know if it is accessing the same address without instrumenting its loads and stores?
• Answer:
  • The first thread
    • publishes its sampled address to a globally visible location
  • The second thread
    • compares its sampled address with the globally published addresses and if there is a match ⇒ inter-thread communication, or
    • uses hardware-debug registers (aka watchpoints) to monitor a globally published address
  • Watchpoint traps when the second thread accesses the same address ⇒ inter-thread communication
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.
Design Components: Debug Registers

**Debug Registers:** Filled with a memory address and length. Sends a *trap* when the memory region specified by the address and length is accessed.
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

PMUs

core 0

Debug registers

PMUs

core 1

Debug registers
Design Components: ComDetective

<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>

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

Design Components:
- **ComDetective**
- **PMUs**
  - core 0
    - Debug registers
  - core 1
    - Debug registers
- perf_event
  - sample_handler
- sample_handler
  - perf_event
Design Components: ComDetective

- **Perf_event**
- **Sample Handler**
- **Trap Handler**
- **Trap Handler**
- **Sample Handler**
- **Perf_event**

<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>

A function that runs when **debug register trap** happens

- **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
- Debug registers

PMUs
- core 1
- Debug registers
perf_event is configured so that **loads** and **stores** to be sampled for every **N events**.

### An Example Workflow

<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
**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

Thread T1

- trap_handler
- sample_handler
- perf_event

PMUs

- core 0
- Debug registers

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>

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

**PMUs**

- **core 0**
  - Debug registers

- **core 1**
  - Debug registers
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.

PMUs

Core 0

Debug registers

Core 1

Debug registers

Thread T0

pers_event sample_handler trap_handler

Thread T1

trap_handler sample_handler perf_event

Debug registers

pmus

Core 0

Core 1

Debug registers

pmus

Core 0

Core 1

Debug registers
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’.

<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>

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.
Another store sample happens on address M3.

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

PMUs

- core 1

Debug registers

Debug registers
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>
</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**
Evaluation

- Accuracy verification with micro-benchmarks
  - Communication volume
  - True/false sharing ratio (reported only in our paper)
  - Point-to-point communication ratio
  - Read/write communication volume (reported only in our paper)
- Communication matrices of large benchmarks
  - 12 PARSEC and 6 CORAL applications
- Use cases: code refactoring
- Sensitivity Analysis (reported only in our paper)
  - Sampling interval impact
  - Debug register count
  - Hash table size
Communication Volume Verification

- Communication volume verification microbenchmark
  - Each thread performs **only store operations** to either **shared data** or **private data** depending on **sharing fraction parameter**.

```c
#pragma omp parallel shared(sharedData) private(privateData) \
  num_threads(nThreads)
{
    for(int i = 0; i < N_ITER; i++) {
      int rNum = rand_r(); // thread private
      If (rNum < SHARING_FRACTION) {
        sharedData = rNum;
      } else {
        privateData = rNum;
      }
    }
}
```

Listing 1: Write-Volume Benchmark
● ComDetective count vs RFO (request for ownership) count when 2-16 threads are mapped to 2 sockets.
● Each thread **only performs store operations to shared data**.
  ○ Real total communication count (ground truth) is RFO count
Point-to-point Communication Ratio Verification

- Point-to-point communication microbenchmark
  - Enables selection of threads which communicate in pairs.
  - In all cases, thread 0 communicates with thread 1 and thread 2 communicates with thread 3.
Snapshot of PARSEC Matrices (only 6 shown)
Snapshot of PARSEC Matrices (only 6 shown)

- **Blacksholes**, financial analysis benchmark
- Splits the price options among threads where each thread can process the options independently from each other
CORAL Benchmarks

AMG

MiniFE

PENNANT

QuickSilver

VPIC

Figure 4: Communication matrices of CORAL benchmarks. Darker color indicates more communication.
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
Summary

A practical tool for capturing inter-thread communication

- Low overhead: 1.27x runtime and 1.3x memory
- High accuracy
- Ability to quantify communication
- Ability to distinguish true vs. false sharing
- Attribute communication to program objects

[Available for download: https://github.com/comdetective-tools]
ParCoreLab@ Koç

http://parcorelab.com

Optimization of Sparse Solvers

Detecting thread communication

Quantum computing

Accelerated Deep Learning

GPU Communication Optimization


