ComDetective: A Lightweight Communication Detection Tool for Threads

Muhammad Aditya Sasongko
Koç University, Istanbul, Turkey
msasongko17@ku.edu.tr

Milind Chabbi
Scalable Machines Research, USA
chabbi.milind@gmail.com

Palwisha Akhtar
Koç University, Istanbul, Turkey
pakhtar19@ku.edu.tr

Didem Unat
Koç University, Istanbul, Turkey
dunat@ku.edu.tr

ABSTRACT

Inter-thread communication is a vital performance indicator in shared-memory systems. Prior works on identifying inter-thread communication employed hardware simulators or binary instrumentation and suffered from inaccuracy or high overheads—both space and time—making them impractical for production use. We propose ComDetective, which produces communication matrices that are accurate and introduces low runtime and low memory overheads, thus making it practical for production use.

ComDetective employs hardware performance counters to sample memory-access events and uses hardware debug registers to sample communicating pairs of threads. ComDetective can differentiate communication as true or false sharing between threads. Its runtime and memory overheads are only 1.30× and 1.27×, respectively, for the 18 applications studied under 500K sampling period. Using ComDetective, we produce insightful communication matrices for microbenchmarks, PARSEC benchmark suite, and several CORAL applications and compare the generated matrices against MPI counterparts. Guided by ComDetective we optimize a few codes and achieve up to 13% speedup.

KEYWORDS

multicore systems, inter-thread communication, communication matrix, hardware performance counters

ACM Reference Format:
Muhammad Aditya Sasongko, Milind Chabbi, Palwisha Akhtar, and Didem Unat. 2019. ComDetective: A Lightweight Communication Detection Tool for Threads . In *Proceedings of ACM Supercomputing (SC'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Inter-thread communication is an important performance indicator in shared-memory multi-core systems [39]. Thread communication information offers valuable insights: it divulges, to an extent, the inner workings of the program without having to examine the code meticulously; it can be used for identifying possible sources of communication-related performance overhead in parallel applications [7, 34]; it can also be used for verifying the multicore hardware design. Therefore, identifying which groups of threads communicate in what volume and their quantitative comparison against expectations offer avenues to tune software for high performance.

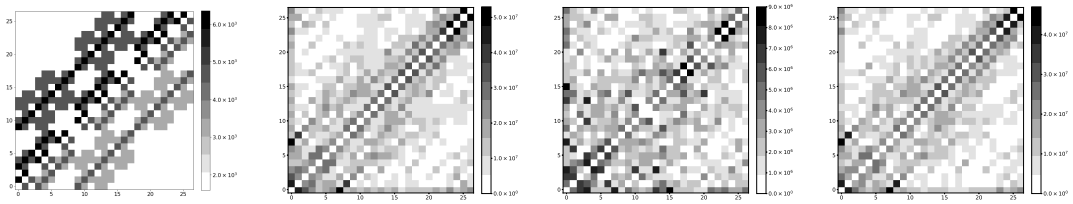
Several techniques exist to capture communication patterns in multi-threaded applications [3, 4, 9, 11, 14, 15, 36]. Though the proposed techniques succeed in generating communication patterns (often called as communication matrix), they come with several limitations. Simulator-based methods (e.g., [4] [11]) (a) make simplistic assumptions about CPU features (e.g., an in-order core), cache protocols and memory hierarchies, (b) introduce $\sim 10,000\times$ runtime slowdown, and (c) generate enormous volume of execution traces that grow linearly with execution time; hence, they are a misfit for evaluating a complex, long-running application in its entirety. Furthermore, to extract communication patterns from simulators, post-mortem analysis of execution traces is needed, which adds additional effort to the user.

Approaches in [36][3][9] use either a modified operating system kernel or hardware extensions to mitigate overheads. The communication pattern that they generate, however, might contain *false communication*[1]—a situation where a cache line that is already evicted by a core is accessed by another core. Such false communication is reported when the accesses to the same cache line by different cores are separated in time. Prior approaches using binary instrumentation techniques, such as [14][15], detect communications only by retaining the thread ids of previous accesses but disregard the timestamps of those accesses. Hence, these schemes also suffer from false communication. An additional source of inaccuracy in binary instrumentation is the time dilation caused by fine-grained instrumentation—the time gap between consecutive accesses by the same core to the same cache line is widened due to the online analysis overheads, which allows other threads to interleave, which in turn results in overestimating communication compared to uninstrumented execution. For example, Numalize [15], one such tool that we use for comparison in our experimental study, dilates execution, changes the execution behavior, and as a result, overestimates total communication count. Other works by Mazaheri et. al [26][27] instrument program code by using a compiler-assisted tool. The code instrumentation enables detection of read-after-write (RAW) and read-after-read (RAR) dependencies among threads in the program and generates true communication (RAW) and reuse (RAR) matrices as outputs. However, their method still introduces large overhead, on average 140× slowdown.

In this work, we propose ComDetective, a communication matrix extraction tool that avoids the drawbacks of the prior art. The

SC'19, Nov 11-16, 2019, Denver, CO
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] False communication should not be confused with false sharing. False sharing results in communication at the hardware level that was not intended by the programmer, while false communication does not lead to inter-core communication.



(a) LULESH - MPI **(b) LULESH** **(c) LULESH True Sharing** **(d) LULESH False Sharing**

Figure 1: Communication matrices of LULESH (Left to Right: MPI, ComDetective: All, True and False Sharing). Darker color indicates more communication.

key premise of ComDetective is to observe the execution with minimal perturbation. ComDetective resorts to the data offered by hardware Performance Monitoring Units (PMUs) and debug registers as a means of measuring inter-thread communication. Hardware PMUs enable extracting the effective addresses involved in loads and stores in sampling fashion. Additionally, debug registers enable monitoring memory access to a designated address by a thread, without introducing any overhead in the intervening window of execution. By employing both PMUs and debug registers, we are able to detect memory accesses performed by different threads on shared cache lines in a short time window while not becoming a severe victim of false communication, unlike other approaches.

Besides being lightweight, ComDetective differentiates communication as true vs. false sharing, where true refers to the actual communication intended by the programmer due to the shared objects and false refers to the false sharing between two threads due to the cache line sharing. Two-dimensional matrices that are generated by tools such as Numalize[14][15] do not differentiate different types of communication. Figure 1 shows a motivating example, where we present the communication matrices for the multi-threaded implementation of LULESH [19] and compare it against the MPI implementation. The MPI matrix is generated using EZTrace [37] and requires post-mortem analysis. Meanwhile executing the application with ComDetective took only 136 sec with 1.48× runtime overhead. In addition, ComDetective can optionally attribute communication to each object in the application. To the best of our knowledge, there exists no other tool for multi-threaded applications that delivers these features while maintaining a low overhead. Our contributions can be summarized as follows:

- ComDetective, a communication detection algorithm and its lightweight tool for multi-threaded applications with the feature to distinguish false vs. true sharing communication
- A thorough evaluation of accuracy, sensitivity, and overhead of ComDetective, and tool's comparison with ground truth and prior work
- Insightful communication matrices of PARSEC benchmark suite and six CORAL applications (AMG, LULESH, MiniFE, PENNANT, Quicksilver, and VPIC), and comparison with MPI communication matrices for the CORAL applications
- Independent of code size, only 30% runtime and 27% memory overheads on the 18 applications studied, making it a practical tool for production use.

The ComDetective tool is publicly available at https://github.com/comdetective-tools.

2 BACKGROUND

Inter-thread communication: We define communication among threads as the transfer of cache lines across different CPU cores due to cache coherence protocol in a shared-memory system. An example is a transfer of cache line from a thread running on a core that has a cache line with 'modified' status, according to MESI protocol, to another thread running on a different core that has the same cache line in the 'invalid' status. Such communication or cache line transfer can also happen from a core that has a cache line with 'exclusive', 'modified', or 'shared' status to another core that does not have that cache line in its local caches.

This kind of communications can occur due to either *true sharing* or *false sharing*. True sharing happens when two different threads communicate or transfer a cache line as both of them access the same variable located in the cache line. False sharing ensues when two threads communicate on a cache line, yet they do not access the same variables, but these variables happen to reside on the same cache line. While true sharing is an inevitable communication for cooperating threads in parallel programs, false sharing can be considered as an overhead since the two threads do not actually need to communicate as they access different variables.

Communication Matrix: Communication matrix is defined as a matrix that counts instances of communications between each pair of threads in a multi-threaded application. The $(i, j)^{\text{th}}$ entry in the matrix represents the number of communication instances between thread i and thread j. The communication matrix is symmetric (both parties are involved in communication) and has zero along the diagonal (a thread does not communicate with itself). The cells only count the number of cache line-granularity data transfers; they do not account other transactions that may be involved by the underlying implementation of the coherence protocol.

Hardware Performance Monitoring Unit (PMU): CPU's PMU offers a programmable way to count hardware events such as loads, stores, CPU cycles, etc. PMUs can be configured to trigger an overflow interrupt once a threshold number of events elapse. A profiler, running in the address space of the monitored program, handles the interrupt and records and attributes the measurements to their corresponding communication types or objects. We refer to a PMU interrupt as a "sample." PMUs are per CPU core and virtualized by the operating system for each OS thread. Intel's Precise Event-Based Sampling (PEBS) [17] facility offers the ability to inspect the effective address accessed by the instruction on an event overflow for certain kinds of events such as loads and stores. This ability to extract the effective address is often referred to as *address*

sampling, which is a critical building block of ComDetective. Such capability has been available in AMD processors via Instruction-Based Sampling (IBS) facility [16] since AMD Family 10h Processors, in POWER processors via Marked Events facility [35] since POWER 5, and in Intel processors via PEBS in Intel Nehalem and their successors.

Hardware debug registers: Hardware debug registers [18, 28] enable trapping the CPU execution for debugging when the program counter reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers with different addresses, widths, and conditions (e.g. `W_TRAP` and `RW_TRAP`) that cause the CPU to trap on reaching the programmed conditions. Today's x86 processors have four debug registers.

Linux perf_events: Linux offers a standard interface to program and sample PMUs and debug registers using a system the `perf_event_open` [21] system call and the associated `ioctl` calls. The ability to program debug registers has been available since Linux 2.6.33, and the ability to access multiple PMUs since Linux 2.6.39 [21]. The Linux kernel can deliver a signal to the specific thread whose PMU event overflows or debug register traps. The user code can (1) `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample and (2) extract the signal context on each debug register trap.

3 COMDETECTIVE

3.1 Overview

In generating communication matrices, ComDetective leverages PMUs and debug registers to detect inter-thread data movement on a sampling basis. If communication is frequent, the same addresses appear in the samples taken on communicating threads; by comparing the addresses seen in closely taken samples on different threads, one can potentially detect communication. If communication is infrequent, however, the probability of seeing the same address in two samples taken by two different threads becomes rare. Hence, ComDetective leverages debug registers to identify infrequent communications. A thread sets a watchpoint for itself to monitor an address recently accessed by another thread. If and when the thread accesses such address in the near future, the debug register traps and thus detects communication.

In ComDetective, each application thread uses PMU to sample its memory access (load and store) events. When a threshold number of events of a certain type (load or store) happen, the corresponding PMU counter overflows. The thread, say T_1, encountering an overflow extracts the effective address involved in the instruction at the time of the overflow (aka sample) and tries to publish the address on to a global data structure, `BulletinBoard`, that other threads can readily access. When another thread, say T_2, encounters its PMU overflow, it looks up the `BulletinBoard` for an address conflicting with its sampled address located on the same cache line. If such an entry is found in `BulletinBoard` and the two accesses are by different threads, then communication is detected between the two threads. If, however, no conflicting entry is found, it may mean the sampled address may be a private address (which is common when the fraction of sharing is less) or it may access it in the near future. In this situation, T_2 picks an *unexpired* address $\mathcal{M}$ posted in

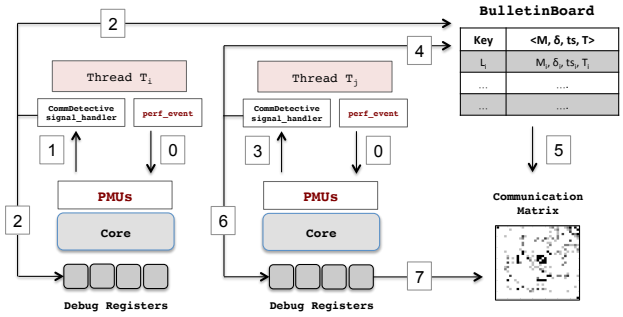


Figure 2: One possible execution scenario: 0) Every thread configures its PMU to sample its stores and loads. 1) Thread T_i's PMU counter overflows on a store. 2) T_i publishes the sampled address to BulletinBoard if no such entry exists and tries to arm its watchpoints with an address in the BulletinBoard (if any). 3) Thread T_j' PMU counter overflows on a load. 4) T_j looks up BulletinBoard for a matching cache line. 5) If found, communication is reported. 6) Otherwise, T_j tries to arm watchpoints. 7) T_j accesses an address on which it set a watchpoint, the debug register traps, communication is reported.

`BulletinBoard` and arms its CPU's debug registers to monitor all or as many as possible addresses that fall on the same cache line $\mathcal{L}$ shared by $\mathcal{M}$. A subsequent access by T_2, anywhere on $\mathcal{L}$, is a communication between T_2 and the thread that published $\mathcal{M}$. This communication will be detected by trapping of the watchpoints in T_2. Once communication is detected, the corresponding communication matrices are updated. The communication is reported if and only if at least one store operation is involved.

ComDetective maintains `BulletinBoard` as a concurrent hash table. The sampled address, rounded down to the nearest cache line address, serves as the key to the `BulletinBoard`; the value for each entry in the `BulletinBoard` is the following tuple: Memory address $\mathcal{M}$ accessed at the point of PMU sample, access length δ, ID of the publishing thread, timestamp of the publishing. Only addresses involved in store operations are inserted into the `BulletinBoard`, but PMU address samples generated for both loads and stores are looked-up in the `BulletinBoard` to detect communication. This arrangement detects both write-after-write and read-after-write sharing; note that any repeating write-after-read sharing in one thread will be captured as a read-after-write sharing in another (the reader) thread.

3.2 Communication Detection Algorithm

The main components of ComDetective and one possible workflow scenario are displayed in Figure 2. Next, we explain the algorithm used in ComDetective.

Setup: Every thread configures its PMU to monitor its memory store and load events. Each of these threads is interrupted on elapsing a specified number of events.

On A PMU Sample: When a PMU counter overflows, the thread T_1 that encounters the overflow, tries to publish the address M_1 that it sampled to `BulletinBoard` and calls PMUSampleHandler presented in Algorithm 1. In Line 6, the thread queries the `BulletinBoard` by using the base address of the cache line L_1 containing M_1. If no entry is found, it tries to arm its watchpoints (WPs) Line 8). If the previously armed WPs are old, the thread T_1 selects an unexpired address M_3 in the `BulletinBoard` and arms its debug registers to monitor the cache line that M_3 belongs to

Algorithm 1 Communication Detection

```
1: global ConcurrentMap BulletinBoard
2: thread_local Timestamp t_prev = 0
3:
4: procedure PMUSampleHandler(Address M_1, AccessLen δ_1, Timestamp ts_1, ThreadID T_1,
   AccessType A_1)
5:     L_1 = getCacheline (M_1)
6:     entry = BulletinBoard.AtomicGet (key=L_1)                    ▷ Is L_1 in hash?
7:     if entry == NULL then                                        ▷ Matching cache line is not found in hash
8:         TryArmWatchpoint(T_1)
9:     else
10:        < M_2, δ_2, ts_2, T_2 > = getEntryAttributes (entry)
11:        if T_1 != T_2 and ts_2 > t_prev then                     ▷ A new sample from a different thread
12:            if [M_1, M_1 + δ_1) overlaps with [M_2, M_2 + δ_2) then
13:                Record true sharing
14:            else
15:                Record false sharing
16:            end if
17:            t_prev = ts_2
18:        else
19:            TryArmWatchpoint (T_1)
20:        end if
21:    end if
22:    if (A_1 is not STORE) or (entry != NULL and M_2 has not expired) then
23:        return
24:    end if
25:                                  ▷ A_1 is a store and the current entry has expired, then publish M_1
26:    BulletinBoard.TryAtomicPut(key = L_1, value = <M_1, δ_1, ts_1, T_1>)
27: end procedure
28:
29: procedure TryArmWatchpoint(ThreadID T)
30:    if current WPs in T are old then
31:        Disarm any previously armed WPs
32:        Set WPs on an unexpired address from BulletinBoard that is not from T
33:    end if
34: end procedure
```

(Line 29-34). Since WPs of a thread belong to the same cache line, they are either all expired or all recent. On x86 with four 8-byte length debug register, ComDetective can monitor only 32 bytes out of the 64 bytes of a cache line. Hence, ComDetective randomly chooses four chunks of the 64-byte cache line to monitor.

In case the entry is already filled by a cache line L_2 from a previous sample and the cachelines are the same, then Line 11 checks the IDs of the publisher thread T_2 and the sampling thread T_1. If thread IDs are different, then communication is detected between T_1 and T_2 (Line 12-16). The communication could be a true sharing or false sharing. If the sampled access region $[M_1, M_1 + \delta_1)$ overlaps with the access region published in `BulletinBoard` $[M_2, M_2 + \delta_2)$ we treat it as a true sharing event and treat it as false sharing event otherwise. We defer the details of how the volume of communication is computed to Section 3.3.

In order not to overcount communications associated with the same published address between two threads, we keep t_{prev} per thread, which is set when a communication is detected for that thread. Line 17 sets t_{prev} to the timestamp of the publisher thread, ensuring that we do not overcount the cache line transfer between two threads. If no communication is recorded for T_1, T_1 tries to arm its WPs Line 19 using an unexpired addressed published by some other thread into the `BulletinBoard`, as described previously.

If either the sample is for a memory load operation or the previously published entry by the same thread is not expired yet, the thread simply returns and resumes its execution. Otherwise, the thread T_1 publishes the sampled address along with other attributes associated with the cache line L_1, such as the timestamp of sampling, memory access length, and thread ID (Line 26). Atomic operations that perform load and store are treated as store.

On watchpoint trap: When a thread T_i experiences a trap in one of the debug registers, T_i is considered to communicate the thread T_j—the thread that had published an address in the `BulletinBoard` whose cache line T_i is monitoring via its debug registers.

After watchpoint trap: After handling the watchpoint trap, the trapping thread disables all debug register armed to monitor the same cache line. This is justified because the subsequent accesses to the same cache line are *expected* to be served locally without generating any communication. If the cache line were modified by another core in the meantime, it will not be detectable and it is indeed not necessary in the coarse-grained sampling scheme. Watchpoints are re-armed with newer published addresses upon next PMU counter overflow, as explained previously.

On program termination: The profiled data need not leave the matrix symmetric. For example, the reported communication may be more in the thread $\langle T_i, T_j \rangle$ pair compared to the thread $\langle T_j, T_i \rangle$ pair. However, since both parties are equally involved in a communication event, we update every $\langle T_i, T_j \rangle$ pair to be the sum of both $\langle T_i, T_j \rangle$ and $\langle T_j, T_i \rangle$, thus making the matrix symmetric.

Expiration period: For practical considerations, each thread treats the timestamp of a `BulletinBoard` entry as "recent" (aka "unexpired") if it was published between its current sample and its previous sample (i.e., one sample period), and "old" (aka "expired") otherwise. This scheme allows each published address or watchpoint to survive long enough to be observed by all threads working at the same rate and yet be naturally evicted by a newer address. A published address is deemed expired, if it survived for more than two store events from the same thread. Load events are not used for determining the expiration period of a published address, since only stores can ever be published into the `BulletinBoard`. The expiration period of watchpoints includes loads as well because watchpoints can be armed by samples generated by loads or stores.

3.3 Quantifying Communication Volume

There are two sources leading to underestimation in communication volume: sparsity of PMU samples and limited number of debug registers to monitor an entire cache line. For instance, four debug registers can cover 32 bytes of the total 64 bytes of an x86-64 cache line. To address the first problem, on each communication detection or trap, instead of recording just one communication event, ComDetective scales up the quantity by the *sampling_period*. In case a communication is detected in a sample and without using debug registers, we update the $Matrix[T_i, T_j]$ cell as: $Matrix[T_i, T_j] \mathrel{+}= sampling\_period$.

To address the second problem, we use the probability theory. If D number of debug registers can monitor M bytes of memory each, they can monitor a total of $D \times M$ bytes. If the CPU cache line is L bytes long, where $L > (D \times M)$, then the probability of trapping on an address involved in a communication after sampling it is $p = (D \times M)/L$. If K traps are detected, in expectation, we can scale it up by $1/p$ to get an estimated number of events, i.e., K/p. Taking both effects into account, on each watchpoint trap, we update the $Matrix[T_i, T_j]$ cell as:

$$Matrix[T_i, T_j] \mathrel{+}= \frac{sampling\_period \times L}{(D \times M)}$$

3.4 Implementation

We implement ComDetective atop the open-source HPCToolkit performance analysis tools suite [1]. ComDetective's profiler loads the monitoring library into the target application's address space at link time for statically linked executables or at runtime using `LD_PRELOAD` [30] for dynamically linked executables. As the target application executes, the profiler in ComDetective manages PMUs and debug registers to record communication pairs. On Intel processors, we use `MEM_UOPS_RETIRED:ALL_STORES` and `MEM_UOPS_RETIRED:ALL_LOADS` to sample memory access events. These events offer the effective memory address accessed in a sample along with the program counter. On a PMU sample, the profiler walks the sampled thread's call stack via an online binary analysis. It, then, attributes the measurements to the sampled call path.

Monitoring stack addresses in the target application is tricky, because the frames of ComDetective's sample/trap handler can overwrite the stack location and cause undesired debug register trap. We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [22]. The `sigaltstack` facility allows each thread in a process to define an alternate signal stack in a user-designated memory region. We use alternate stack to handle PMU and watchpoint signals. All other signals continue to use the default stack unless specified otherwise by the application.

ComDetective optionally allows mapping each communication event to runtime objects in the program. It uses ADAMANT[8] to extract static and dynamic object information. Static objects are detected by parsing the binary file and the dynamic objects are detected by intercepting allocation routines such as `malloc` and `free`. All stack objects of a given thread are grouped into a single object, while dynamic objects that have the same call stack are grouped into an object.

4 EXPERIMENTAL STUDY

This section evaluates the accuracy, sensitivity, and overheads of ComDetective and presents insightful communication matrices for the selected CORAL and PARSEC benchmarks. Our evaluation system is a 2-socket Intel Xeon E5-2640 v4 Broadwell CPU. There are 10 cores per socket with 2-way simultaneous multi-threading. Each core has its own local L1i, L1d, and L2 caches, while all cores in a socket share a common L3 cache. We use `Linux 4.15.0-rc4+` and `GNU-5.4` toolchain. Unless otherwise stated, the default sampling interval in all experiments is 500K for both reads and writes and the default hash table size in `BulletinBoard` is 127.

4.1 Accuracy Verification

We evaluate the accuracy of ComDetective with four microbenchmarks we have developed. These benchmarks assess the accuracy against the known ground truth by varying the parameters such as communication volume, false sharing fraction, communicating thread subgroups, and read-to-write ratios.

4.1.1 ***Write-Volume.*** In this benchmark, each thread performs only a single store operation (atomic write) in each iteration of a loop as shown in Listing 1. Each thread randomly either accesses its private data or common shared data. The ratio of accesses to shared vs. private data is controlled via the `SHARING_FRACTION`. For example, if the sharing fraction is specified as 20%, then approximately 20% of the time over the entire execution, thread writes into the shared data and writes to its private data in the remaining 80% of the time. There is no false sharing in this benchmark. The source of ground truth for this benchmark is the sum of L2_RQSTS.ALL_RFO hardware performance event obtained from each thread in the absence of other cache sharing effects (which there is none in the benchmark). An RFO event happens when a core tries to gain ownership of a cache line for updating it.

Figure 3 displays the results with different number of threads for the *Write-Volume* benchmark, where the x-axis is the sharing fraction and y-axis is the total communication volume. Figure 3-a and b, respectively, show thread mapping to the same socket (compact)

```
#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

```

#pragma omp parallel shared(trueSharingData, falseSharingData) \
  private(privateData) num_threads(nThreads)
{
 int tid = omp_get_thread_num();
 atomic<uint64_t> * falseShared = &(falseSharingData[tid]);
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < FALSE_SHARING_FRACTION) {
      *falseShared += rNum;
    } else {
      trueSharingData += rNum;
}}}
```

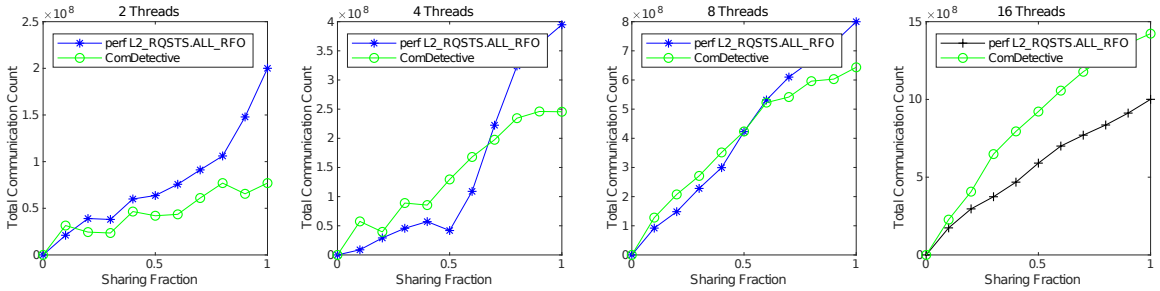
Listing 2: False Sharing Benchmark

```
#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 < READ_FRACTION) {
      rNum = sharedData;
    } else {
      sharedData = rNum;
}}}
```

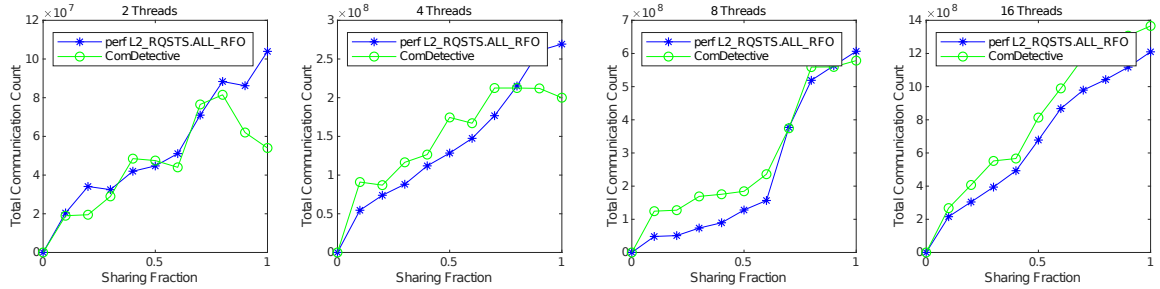
Listing 3: Read-Write Benchmark. Reading from shared data vs. writing to shared data

```
#pragma omp parallel shared(sharedDataArray) private(privateData) \
 num_threads(nThreads)
{
 int tid = omp_get_thread_num();
 int shared_data_index = getSharedDataIndex(tid);
 int sharing_fraction =  getSharingFraction(shared_data_index);
 atomic<uint64_t> * sharedData = \
   &(sharedDataArray[shared_data_index]);
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < sharing_fraction) {
      *sharedData = rNum;
    } else {
      privateData = rNum;
}}}
```

Listing 4: Point-to-point Communication Benchmark. Communication happens between threads that have the same `shared_data_index` value



(a) Total communication counts for across different sharing fractions with threads mapped to a single socket (compact).



(b) Total communication counts for different sharing fractions with threads mapped evenly to two sockets (scatter).
Figure 3

vs. two different sockets (scatter). As expected, the communication volume increases as the sharing fraction increases or thread count increases. However, notice that the actual communication volume collected via RFO does not follow a straight line and in most cases, ComDetective is very accurate in capturing this trend. The nonlinear growth of communication is because when the same cache line is repeatedly accessed by the same core, even if there is a pending request from another core, the request from the core that holds the line is unfairly favored. While such optimizations are not unexpected from a CPU design perspective, they are unintuitive for a programmer and make it harder for them to envision the communication pattern and volume in their programs without the help of tools such as ComDetective. Another unintuitive behavior is that mapping threads to different sockets results in less communication than when they are mapped to the same socket and ComDetective can identify this phenomenon. We have also performed similar experiments with `atomic_add` and `compare_and_swap` and observed similar behaviors.

The gaps of undercounting and overcounting in certain cases is an artifact of sampling that relies on probability theory in estimating total number of communications between any two threads. As described in Sec 3.3, we use sampling period to estimate the number of communication events that might have been missed between samples. Because of this reason, certain degree of undercounting and overcounting with respect to the ground truth is inevitable.

In Figure 3-a and b, ComDetective underestimates the number of communications when the thread count is small and the sharing fraction is high (~100%). This undercounting can be attributed to signal handling. When a thread (say T_1) takes a PMU sample or watchpoint trap, T_1's execution gets diverted to handling the signal. During signal handling, T_1 will not generate any cache line communication with its peer thread (say T_2). During this time, T_2 progresses unhindered and continues performing memory access operations across its loop iterations. The act of monitoring reduces communication and hence it appears as undercounting with respect to the unmodified original execution. Note however that this level of extreme sharing without any computation as in our synthetic benchmark shown in Listing 1 is as a pathological case for ComDetective and unlikely in real-world code.

The right most plot in Figure 3-a presents the communication volume for 16 threads running on 10-core socket, where some of the physical cores are oversubscribed with more than one thread. From the figure, it appears that ComDetective overestimates the communication. However, RFO events are no longer the ground truth in this case. This is because `L2_RQSTS.ALL_RFO` counts RFO events between physical cores at L2 caches; and L2 is shared by logical cores. As a result, communication happening between the threads mapped to the same physical core does not result in an RFO event. The RFO counts of threads sharing a physical core are combined if they communicate with other physical cores. Consequently, one would expect that the RFO counts should be lower than the actual communication count when cores are oversubscribed. Indeed, ComDetective gives higher counts than the counts of `L2_RQSTS.ALL_RFO` events.

We compare ComDetective with the state of the art in Figure 4, which plots the communication volume captured by Numalize [15], ComDetective, and the ground truth when two threads are mapped to the same or different sockets using atomic add benchmark. Numalize hugely overestimates the volume possibly because it does not maintain the timestamp of accesses, records many false communications, and ignore data from the underlying hardware.

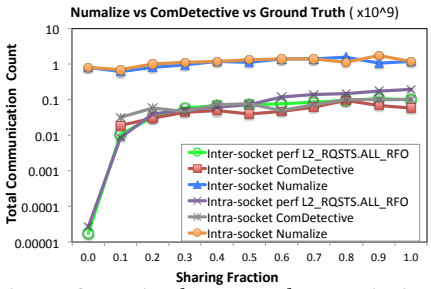


Figure 4: Comparison between total communication counts captured by Numalize[15], ComDetective, and the real RFO counts

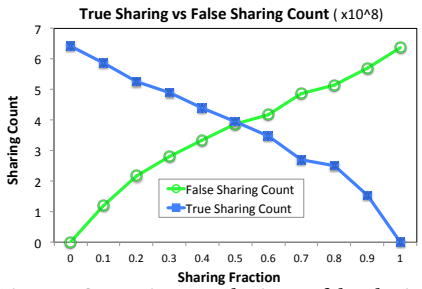


Figure 5: Comparing true sharing vs. false sharing counts across different sharing fractions using 8 threads.

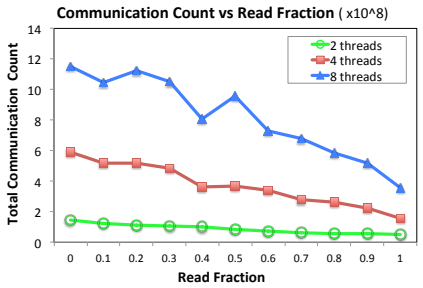


Figure 6: Total communication counts detected by ComDetective across different fraction of read operations.

4.1.2 ***False-Sharing.*** Unlike *Write-Volume*, which has no false sharing, this benchmark introduces a controllable amount of false sharing as shown in Listing 2. Also for coverage, instead of an atomic write, it performs atomic add operation. This benchmark is valuable to assess the statistical nature of randomly selecting parts of a cache line to observe using limited number of debug registers. The ratio of false sharing to the entire communications captured is expected to match the fraction of false sharing specified by the user. Figure 5 shows the true and false sharing counts for eight threads with varying false sharing fractions. As expected, the false sharing count increases linearly as false sharing fraction increases. Furthermore, the ratio of false sharing count to total communication count is very close to the specified false sharing fraction for each data point.

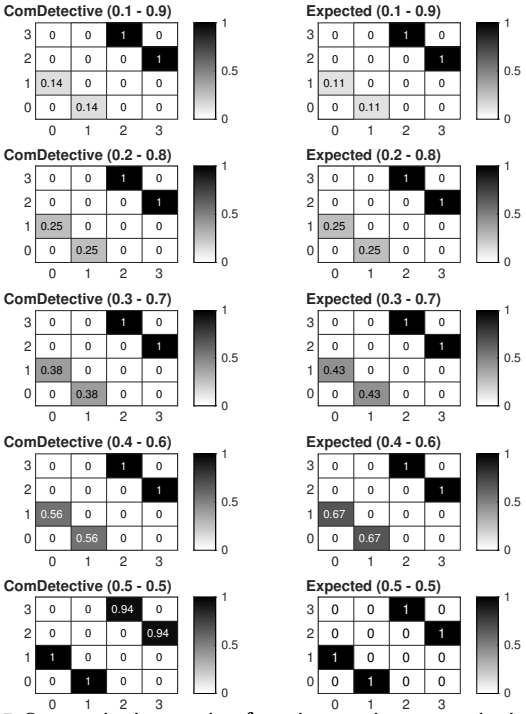


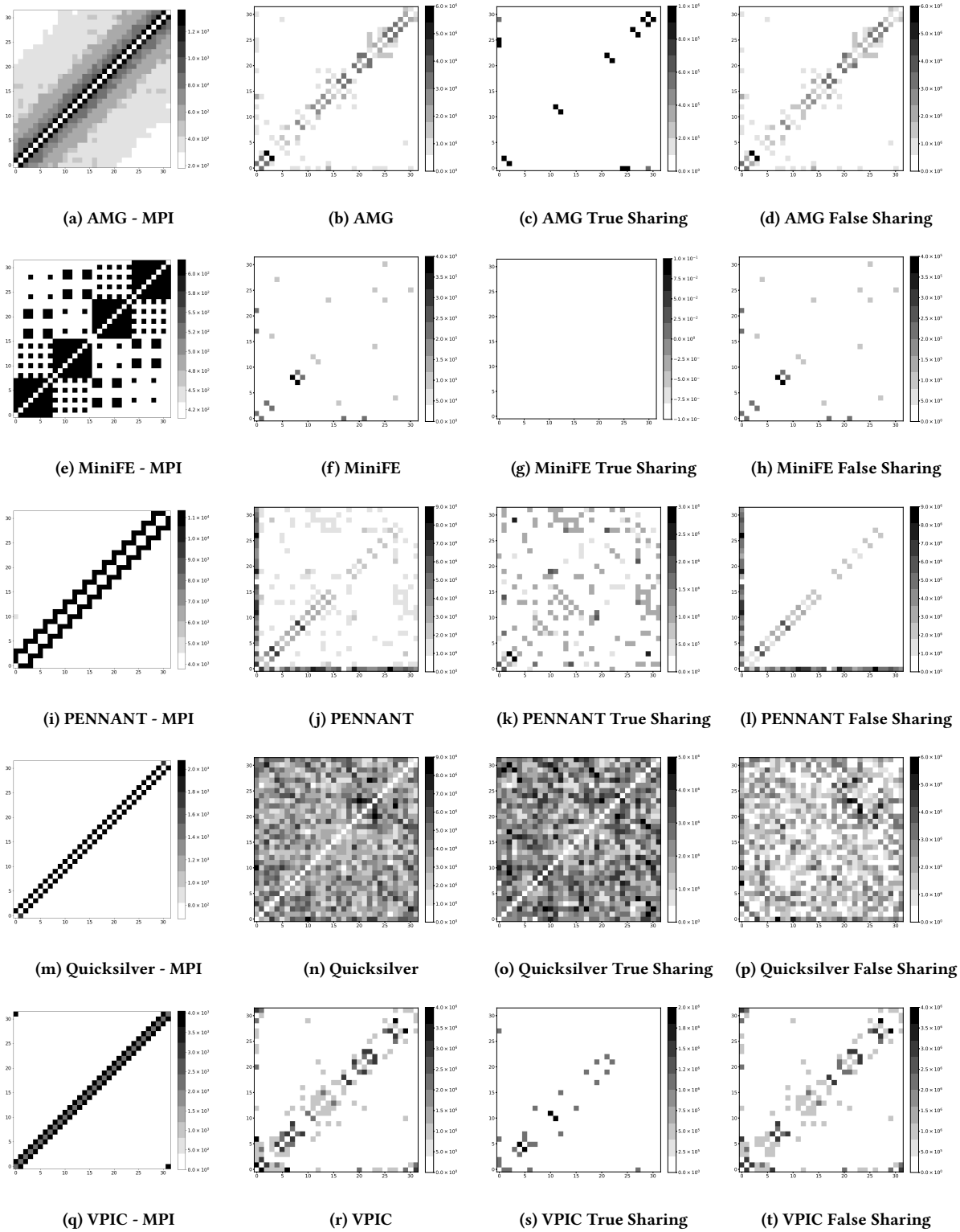
Figure 7: Communication matrices for point-to-point communications having different sharing fractions. Thread 0 only communicates with thread 1, thread 2 only communicates with thread 3. Sharing fractions for each pair are shown on the top of the maps.

4.1.3 ***Read-Write.*** Since only store operations are inserted into the `BulletinBoard`, it is important to assess the quality of results for benchmarks that involve a mix of loads and stores. The benchmark is configured so that one thread always and only performs a write operation in each iteration in a shared location, while the remaining threads might perform either a write or a read operation on the same shared data depending on the specified read fraction. The usage of the read fraction to control the amount of read operations is illustrated in Listing 3. For the compiler not to eliminate the loads, the loads are implemented with asm volatile. As read fraction increases, more and more reads hit in the local cache before the newly written value by the writer are visible. Thus, increasing the reading fraction linearly decreases the communication volume. Figure 6 captures the total detected communication count as a function of read fraction at different thread counts (2, 4, and 8). The communication volume is naturally higher when there are more number of readers. It is worth noting that the drop in communication is more steep with increasing reading fraction for larger number of threads than for a fewer number of threads.

4.1.4 ***Point-to-Point Communication.*** In this benchmark, threads are grouped in pairs and the shared variables are per pair instead of a single shared variable for all threads. This benchmark evaluates the accuracy of point-to-point communication (every cell of the communication matrix). To make a pair of threads communicate, they both need to have similar values of index variables (`shared_data_index`), which point to a same shared array element that they write into as shown in Listing 4. Figure 7 shows the results for two groups performing only write operations; thread 0 communicates only with thread 1, and thread 2 only communicates with thread 3. Figure 7 shows the communication matrices as heat maps; the observed communication is on the left side and the expected results are on the right side. The number in each matrix cell displays the *normalized* communication count in that cell, which is computed by dividing each cell by the cell with the highest count in its matrix. It is evident that heat maps produced by ComDetective resemble the expected heat maps.

4.2 Communication in CORAL Benchmarks

In this section, we present insightful communication matrices for the selected CORAL and CORAL-2 benchmarks, namely AMG [2, 41], LULESH [24], miniFE [29], PENNANT [32], Quicksilver [33], and VPIC [6, 40] as heatmaps in Figure 8, where darker color indicates more cache line transfers between pairs. The matrices



(a) AMG - MPI (b) AMG (c) AMG True Sharing (d) AMG False Sharing

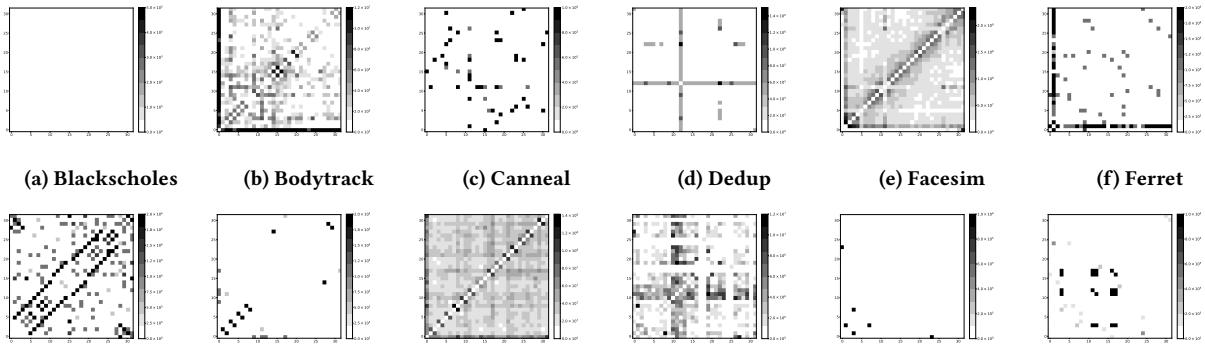
(e) MiniFE - MPI (f) MiniFE (g) MiniFE True Sharing (h) MiniFE False Sharing

(i) PENNANT - MPI (j) PENNANT (k) PENNANT True Sharing (l) PENNANT False Sharing

(m) Quicksilver - MPI (n) Quicksilver (o) Quicksilver True Sharing (p) Quicksilver False Sharing

(q) VPIC - MPI (r) VPIC (s) VPIC True Sharing (t) VPIC False Sharing

Figure 8: Communication matrices of CORAL benchmarks. Darker color indicates more communication.



(a) Blackscholes (b) Bodytrack (c) Canneal (d) Dedup (e) Facesim (f) Ferret

(g) Fluidanimate (h) Freqmine (i) Streamcluster (j) Swaptions (k) Vips (l) x264

Figure 9: Communication matrices of PARSEC benchmark suites. Darker color indicates more communication.

	Execution Time (sec)		Data Movement (GB)	
	MPI	OpenMP	MPI (Msg Size)	OpenMP (Cache Lines)
AMG	35.19	39.22	6.22	7.33
MiniFE	111.82	142.25	3.24	1.46
Quicksilver	19.04	23.45	32.74	106.13

Table 1: Running time and data movement comparison of OpenMP and MPI implementations for AMG, MiniFE and Quicksilver using 32 threads

are core-indexed not thread-indexed as ComDetective can covert the thread IDs to core IDs using the `sched_getcpu()` system call if needed. The threads in each benchmark are bound to the cores with compact mapping strategy but evenly distributed to two sockets.

We compare the inter-thread communication matrices generated by ComDetective with the inter-process communication matrices generated by EZTrace [37]. EZTrace is a generic trace generation framework and it collects the necessary information by intercepting function calls and recording events during execution using the FxT library [12] and then performs a post-mortem analysis on the recorded events. The MPI and OpenMP variants of all six applications are based on the same source distributions with optional flags to turn on/off the OpenMP/MPI compilation in their makefiles. As a result, there are no significant algorithmic differences in their implementations. The MPI matrices report the total number of messages exchanged between processes, not the message size. All applications use 32 threads for OpenMP and 32 ranks for MPI except for LULESH which uses 27 threads (or ranks) since it needs a cubic number. For the hybrid implementations of MPI, we set the thread count per rank to 1.

In general, ComDetective offers insights into communication patterns in these applications. For example, the following patterns emerge from our matrices: 1) L-shape pattern in the lower left corners (e.g. *LULESH, PENNANT*), which indicates that all threads heavily communicate with the master thread (a central bottleneck), 2) nearest neighborhood communication pattern, where threads mostly communicate with adjacent threads (e.g. *AMG, MiniFE, VPIC*), and 3) group communications (e.g. *Quicksilver, LULESH*). Although the inter-thread communication matrices are generally more populated than the inter-process communication matrices, in most cases, they logically resemble their MPI counterparts except for MiniFE and Quicksilver. Quicksilver uses a mesh in its computation and the user defines mesh elements per dimension. If the decomposition geometry is not explicitly specified by the user for the MPI ranks, the MPI communication matrix (not shown) becomes very similar to ComDetective's matrix. However, following the suggested decomposition by the Quicksilver developers [33] we decompose the mesh in only one dimension, resulting in nearest neighborhood communication for MPI. It is not possible for a user to perform similar type of decomposition for threads in a configuration file, resulting in more neighbors to communicate.

The total communication counts captured by the communication matrices might help explain the performance difference between OpenMP/MPI versions and scalability of benchmarks. Table 1 presents the execution time of the *AMG, MiniFE* and *Quicksilver* applications. The table also shows the resulting data movement for each benchmark, where data movement for the multi-threaded applications is calculated based on the total number of cache line transfers in Gbytes with the help of ComDetective. Similarly, for MPI, we computed the total message size exchanged including peer-to-peer and collective communications with the help of EZTrace. In all three applications, MPI outperforms OpenMP. This result, perhaps, can be attributed to the fact that the MPI implementations lead to less data movement than their OpenMP counterparts. For example, the multi-threaded versions of *AMG* and *Quicksilver* perform respectively 11% and 23% more data movement than the multi-process versions. The exception for this is *MiniFE*, in which the communication count of its OpenMP implementation is lower than its MPI counterpart. However, while the MPI version exchanges 0.5M messages for its data movement, the OpenMP version of *MiniFE* leads to 24.5M cache line transfers during its execution, which explains the performance gap.

Figure 8 also splits the inter-thread communication matrices into two matrices one each for true and false sharing. Due to the space limitation, we discuss the false sharing matrices for only MiniFE, which solves kernels of finite-element applications. It generates a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements and then solves the linear-system using a conjugate-gradient algorithm. ComDetective shows that the communication is among the adjacent threads (other than with the thread id 0) and dominated by false sharing. False sharing occurs `sum_in_symm_elem_matrix` and `sum_into_vector` functions, where adjacent elements in a vector falling into a single cache line are accessed by different threads.

While padding each scalar forming the elements of a vector can eliminate such false sharing, it can also have the deleterious effect of bloating the memory.

4.3 Communication in PARSEC Benchmarks

Figure 9 shows the PARSEC matrices created by ComDetective. Our matrices differ from the ones previously studied by [4], [10] and [15]. In general, ours are sparser. This can be explained by the fact that our approach takes into account the cache coherency protocol. Since we use expiration period to discard false communications among threads, which might happen due to the huge time gap between memory accesses by two supposedly communicating threads, our tool records much fewer false positives than the techniques previously used. In fact, ComDetective identifies no communication for *Blackscholes* and very infrequent communication for *Vips* and *Freqmine*. *Blackscholes* and *Vips* indeed exhibit very low communication, which is also pointed out by the PARSEC authors [5]. For example, *Blackscholes*, which is a financial analysis benchmark, splits the price options among threads where each thread can process the options independently from each other. Communication can potentially occur at the boundaries of the partitions if boundaries share a cache line. However, it is very unlikely for threads to access the boundaries around the same time because these accesses are far separated in time. The PARSEC authors note that *Freqmine* has a high amount of sharing; however it has a very large working set size too, which implies that accesses are served from memory, not from cache. Moreover, the work in [4] fails to identify any meaningful communication patterns for *Bodytrack, Dedup, Facesim, Ferret, Streamcluster* and *Swaptions*, on the other hand, ComDetective successfully detects these patterns.

4.4 Use-Case: Data Structure Optimization

ComDetective can optionally map detected communications, either true or false sharing, to the data objects that experience them at the expense of slightly increased overhead. Object-level attribution and quantification offers actionable feedback to the developers for object-specific optimizations or code modifications for performance tuning. To demonstrate this feature, we analyzed PARSEC's *fluidanimate* and *streamcluster* to identify their data objects that suffer from false sharing the most. After identifying and analyzing these objects, we modified some of their data structures to reduce false sharing and improve the applications' performance.

For *fluidanimate*, false sharing is caused by several dynamically allocated objects and a global variable named `barrier`. Due to the size of the dynamically allocated objects, applying padding among object elements might result in memory bloat. Therefore, we modified only the data structure of `barrier`. The variable `barrier` is a struct that has `pthread_cond_t` as an attribute. Since the attributes of `pthread_cond_t` are read and written by multiple threads in the `pthread_cond_wait` function, we introduced padding among the attributes of `pthread_cond_t` in the pthread library. After this modification, we achieved 13% speedup in *fluidanimate*.

For *streamcluster*, most of its false sharing is due to inter-thread synchronization by using `pthread_mutex_t` data structure. By introducing padding to the mutex attributes in the pthread library and no changes in *streamcluster* itself, we achieved 6% speedup.

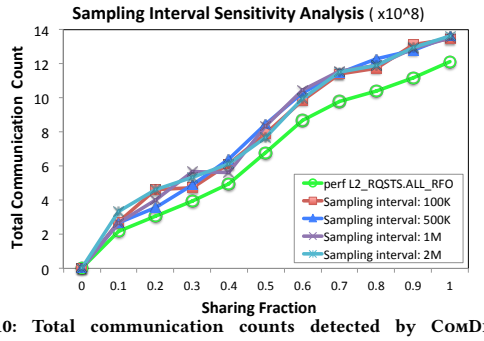


Figure 10: Total communication counts detected by ComDetective under different sampling intervals compared with the ground truth (L2_RQSTS.ALL_RFO counts) when 16 threads are mapped to 2 sockets

Sampling Interval	Runtime Overhead			Memory Footprint Overhead		
	AMG	LULESH	MiniFE	AMG	LULESH	MiniFE
100K	1.07×	2.12×	1.16×	1.00×	1.76×	1.00×
500K	1.10×	1.48×	1.10×	1.00×	1.62×	1.00×
1M	1.07×	1.33×	1.06×	1.00×	1.58×	1.00×
2M	1.08 ×	1.20×	1.03×	1.00×	1.51×	1.00×
	PARSEC + CORAL			PARSEC + CORAL		
500K	1.30×			1.27×		

Table 2: Runtime and space overhead of ComDetective under different sampling intervals for applications using 32 threads (LULESH 27 threads)

4.5 Sensitivity and Overhead Analysis

4.5.1 BulletinBoard Size: To test the sensitivity of the ComDetective under different hash table sizes, we use the *Write-Volume* benchmark but vary the size of `BulletinBoard`. Using 16 threads, we observe no difference in total communication counts detected by ComDetective under hash table sizes of 5, 17, 31, 61 and 127. Furthermore, we evaluate the performance overhead at different hash table sizes using LULESH [19]. Increasing the hash table size does not materially affect the runtime overhead. For that reason, we use 127 as the hash table size for all experiments.

4.5.2 Sampling Interval: We measure the sensitivity of the tool against sampling interval in terms of both the accuracy and overhead using the *Write-Volume* benchmark with 16 threads. Figure 10 shows the total communication counts under different sharing fractions and sampling intervals from 100K up to 2M. The detected total communication count does not deviate much from the ground truth across all sampling intervals. However, we expect that in an application where communication is infrequent, a large sampling interval would result in highly sparse communication matrices or no communication would be detected in the worst case. In such cases, a small sampling interval should be chosen at the expense of increasing overhead.

4.5.3 Overhead: Table 2 displays the performance overhead of ComDetective under different sampling intervals for AMG, LULESH and MiniFE. As seen from the table, the tool has a low space overhead, which allows it to be used in practice for large-scale applications. The runtime overhead drops significantly when the sampling interval is increased from 100K to 500K for LULESH and the overhead is even lower for the other two applications. Since ComDetective maintains good accuracy with reasonable performance overhead on average at a sampling interval of 500K, we

chose 500K as the default sampling interval for all experiments. For the twelve PARSEC benchmarks, the runtime overhead ranges from 1.03× (*streamcluster*) to 2.10× (*x264*) with an average of 1.32×. For the six CORAL benchmarks, the runtime overhead ranges from 1.02× (PENNANT) to 2.17× (VPIC) with an average of 1.27×.

4.5.4 Debug Registers: x86 processors have four debug registers, and ComDetective uses all four for arming watchpoints. We study the impact of the number of debug registers (1, 2, 3 and 4) on the total communication counts detected by ComDetective for 16 threads using the *Write-Volume* benchmark. We observed that the number of debug registers has a negligible impact on the accuracy of ComDetective. This is because when we quantify the communication volume, we scale the volume based on the number of debug registers as discussed in Section 3.3.

5 RELATED WORK

Simulator-based Approaches: Barrow-Williams et al. [4] generate communication patterns for SPLASH-2 and PARSEC benchmarks by collecting memory access traces using Virtutech simics simulator [25]. Thread table of the kernel running on the simulator is also accessed to keep track of all running threads. Similar to [4], Henrique Molina da Cruz et al. [11] also employ a simulator to generate memory access traces. The resulting memory traces are used as the basis to create memory sharing matrix. By considering the memory sharing matrix, thread affinity is implemented by taking memory hierarchy into account. Application threads are mapped according to the generated thread affinity by using Minas framework [31]. ComDetective differs from these techniques in the way that they generate thread communication pattern with the help of a hardware simulator, while we generate communication matrix by PMUs. This makes ComDetective practical to use and runs faster than the simulator-based techniques, especially for full application execution.

OS-based Approaches: Tam et al. [36] and Azimi et al. [3] obtain communication patterns from running parallel applications through PMUs. Unlike ComDetective, their technique requires kernel support. PMUs are accessed by the kernel and the communication pattern of a running application can be generated by the kernel. The PMUs that are accessed are pipeline stall cycle breakdown, L2/L3 remote cache access counters, and L1 cache miss data address sampler.

Cruz et al. [9] use Translation Look-aside Buffers (TLBs) to generate of communication matrix that records page level memory sharing. Two approaches were introduced that use software-managed TLB and hardware-managed TLB. For the software-managed TLB, a trap is sent to OS when TLB miss occurs. Before the missing page table entry is loaded, TLB content of each core is checked for the matches of the missing entry. The information on the matches is used to update the communication matrix. For the hardware-managed TLB, kernel will check the content of TLBs periodically. Both approaches require OS support. In contrast, ComDetective uses user-space PMU sampling. Moreover, TLB-granularity monitoring is too coarse-grained because inter-thread communications happen at cache-line granularity.

Code Instrumentation-based Approaches: Diener et al. [14, 15] develop Numalize, which uses binary instrumentation [23] to intercept memory accesses and identify potential communications among threads by comparing the intercepted memory accesses. Two or three threads that perform accesses to a memory block consecutively are considered to communicate by the tool. We have compared ComDetective with Numalize in our experimental study. Numalize introduces more than 16× runtime overhead and almost 2000× memory overhead, whereas ComDetective introduces only 1.30× runtime overhead and 1.27× space overhead. Moreover, ComDetective does not dilate execution and produces more accurate communication matrices.

A more recent work [26, 27] performs code instrumentation with the help of the LLVM compiler. This instrumentation allows detection of RAW and RAR dependencies in the original code and outputs this information as communication and reuse matrices. Through communication reuse distance and communication reuse ratio derived from these outputs, the tool facilitates analysis of communication bottlenecks that arise from thread interactions in different code regions. However, this tool still suffers from significant slowdown (140×), and is limited to detection of memory accesses to similar addresses. Hence, to our knowledge, it cannot detect cache line transfers that are triggered by false sharing.

Profiling Memory Accesses: Concerning the use of Performance Monitoring Units (PMUs) by library or standalone tool to profile memory accesses or data movement, our work is not the first one that implements this idea. Lachaize et al. [20] introduced MemProf, which utilizes kernel function calls to sample data from memory access events. This data is used to identify objects that are accessed remotely by any thread. Like ComDetective, MemProf also intercepts functions for thread creation, thread destruction, object creation, and object destruction to differentiate memory accesses belonging to different objects and different threads. Unat et al. [38**?**] introduce a tool, ExaSAT, to analyze the movement of data objects using compiler analysis. Even though it has no runtime overhead, it cannot capture all the program objects or their references as it relies on static analysis. Chabbi et al. [7] employ PMUs and debug registers to detect false sharing but do not generalize it for inter-thread communication matrices; furthermore, their technique does not quantify communication volume even for false sharing. Even though these tools can count memory access events, they do not associate these events to threads and are not used in generating communication pattern among threads.

6 CONCLUSIONS

Inter-thread communication is an important performance indicator in shared-memory systems. We developed ComDetective, a communication matrix generation tool that leverages PMUs and debug registers to detect inter-thread data movement on a sampling basis and avoids the drawbacks of prior work by being more accurate and introducing low time and memory overheads. We present the algorithm used by ComDetective and its implementation details, then evaluate the accuracy, performance, and utility of the tool, by carrying out extensive experiments. Tuning code based on the insights gained from ComDetective delivered up to 13% speedup. Programmers can generate insightful communication matrices, differentiate true and false sharing, associate communication to objects, and pinpoint high inter-thread communication in their applications with the help of ComDetective.

ACKNOWLEDGMENTS

The authors from Koç University are supported by the Scientific and Technological Research Council of Turkey (TUBITAK), Grant no. 215E193.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency Computation: Practice Experience* 22, 6 (2010), 685–701.

[2] AMG. 2017. Parallel Algebraic Multigrid Solver. https://github.com/LLNL/AMG.

[3] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. 2009. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 56–65.

[4] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of Splash-2 and Parsec. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[6] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 2008. 0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413435

[7] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False-sharing Detection. In *2018 SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[8] Pietro Cicotti and Laura Carrington. 2016. ADAMANT: Tools to Capture, Analyze, and Manage Data Movement. In *The International Conference on Computational Science, 2016. ICCS 2016.*

[9] Eduardo H.M. Cruz, Matthias Diener, and Philippe O.A. Navaux. 2012. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*.

[10] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2019. EagerMap: A Task Mapping Algorithm to Improve Communication and Load Balancing in Clusters of Multicore Systems. *ACM Trans. Parallel Comput.* 5, 4, Article 17 (March 2019), 24 pages. https://doi.org/10.1145/3309711

[11] Eduardo Henrique Molina da Cruz, Marco Antonio Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and Jean-Francois Mehaut. 2011. Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*.

[12] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. 2005. An Efficient Multi-level Trace Toolkit for Multi-threaded Applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing (Euro-Par'05)*. 166–175.

[13] E. Deniz, A. Sen, B. Kahne, and J. Holt. 2015. MINIME: Pattern-Aware Multicore Benchmark Synthesizer. *IEEE Trans. Comput.* 64, 8 (Aug 2015), 2239–2252. https://doi.org/10.1109/TC.2014.2349522

[14] Matthias Diener, Eduardo H.M. Cruz, Laercio L. Pilla, Fabrice Dupros, and Philippe O.A. Navaux. 2015. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88-89 (2015), 18–36.

[15] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, and Philippe O. A. Navaux. 2016. Communication in Shared Memory: Concepts, Definitions, and Efficient Detection. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

[16] Paul J. Drongowski. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf.

[17] Intel. 2010. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[18] Mark Scott Johnson. 1982. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 140–148. https://doi.org/10.1145/800050.801837

[19] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[20] Renaud Lachaize, Baptiste Lepers, and Vivien Quema. 2012. MemProf: a memory profiler for NUMA multicore systems. In *USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference*. 5.

[21] Linux. 2012. perf_event_open - Linux man page. https://linux.die.net/man/2/perf_event_open.

[22] Linux. 2018. SIGALTSTACK. http://man7.org/linux/man-pages/man2/sigaltstack.2.html.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 190–200.

[24] LULESH 2.0. [n. d.]. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://github.com/LLNL/LULESH.

[25] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[26] Arya Mazaheri, Felix Wolf, and Ali Jannesari. 2015. Characterizing Loop-Level Communication Patterns in Shared Memory Applications. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP 2015)*. https://doi.org/10.1109/ICPP.2015.85

[27] Arya Mazaheri, Felix Wolf, and Ali Jannesari. 2018. Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3225058.3225142

[28] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. 1982. Guidelines for Creating a Debuggable Processor. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 100–106. https://doi.org/10.1145/800050.801833

[29] miniFE. [n. d.]. MiniFE Finite Element Mini-Application. https://github.com/Mantevo/miniFE.

[30] Greg Nakhimovsky. 2001. Debugging and Performance Tuning with Library Interposers. http://dsc.sun.com/solaris/articles/lib_interposers.html.

[31] Dimitrios S. Nikolopoulos, Eduard AyguadÃĺ, and Constantine D. Polychronopoulos. 2002. Runtime vs. Manual Data Distribution for Architecture-Agnostic Shared-Memory Programming Models. *International Journal of Parallel Programming* 30, 4 (2002), 225–255.

[32] PENNANT. 2016. Unstructured mesh hydrodynamics for advanced architectures. https://github.com/lanl/PENNANT.

[33] Quicksilver. [n. d.]. A proxy app for the Monte Carlo Transport Code, Mercury. https://github.com/LLNL/Quicksilver.

[34] Pirah Noor Soomro, Muhammad Aditya Sasongko, and Didem Unat. 2018. BindMe: A thread binding library with advanced mapping algorithms. *Concurrency and Computation: Practice and Experience* 30, 21 (2018). https://doi.org/10.1002/cpe.4692

[35] M. Srinivas, B. Sinharoy, R. J. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, P. Seshadri, J. W. Kellington, A. Mericas, A. E. Petruski, V. R. Indukuru, and S. Reyes. 2011. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD* 55, 3 (May-June 2011), 4:1–4:19.

[36] David Tam, Reza Azimi, and Michael Stumm. 2007. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 47–58.

[37] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra. 2011. EZTrace: A Generic Framework for Performance Analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 618–619. https://doi.org/10.1109/CCGrid.2011.83

[38] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. 2015. ExaSAT: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications* 29, 2 (2015), 209–232. https://doi.org/10.1177/1094342014568690 arXiv:https://doi.org/10.1177/1094342014568690

[39] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[40] VPIC. [n. d.]. Vector Particle-In-Cell (VPIC) Project. https://github.com/lanl/vpic.

[41] Ulrike Meier Yang. 2006. Parallel Algebraic Multigrid Methods High Performance Preconditioner. *Numerical Solution of Partial Differential Equations on Parallel Computers, LNCS 51* (2006), 209–233.