# Using differential execution analysis to identify thread interference

Mohamed Said Mosli Bouksiaa*, François Trahay*, Alexis Lescouet*, Gauthier Voron*†, Rémi Dulong*, Amina Guermouche*, Élisabeth Brunet* and Gaël Thomas*
*Telecom SudParis, †LIP6/UPMC

**Abstract**—Understanding the performance of a multi-threaded application is difficult. The threads interfere when they access the same shared resource, which slows down their execution. Unfortunately, current profiling tools report the hardware components or the synchronization primitives that saturate, but they cannot tell if the saturation is the cause of a performance bottleneck. In this paper, we propose a holistic metric able to pinpoint the blocks of code that suffer interference the most, regardless of the interference cause. Our metric uses performance variation as a universal indicator of interference problems. With an evaluation of 27 applications we show that our metric can identify interference problems caused by 6 different kinds of interference in 9 applications. We are able to easily remove 7 of the bottlenecks, which leads to a performance improvement of up to 9 times.

**Index Terms**—performance analysis, multithreading, bottleneck detection.

## 1 Introduction

Analyzing the performance of a multi-threaded application is important to identify which blocks of code hamper the parallelism. However, performing this analysis is difficult because almost any hardware component and any block of code can become a bottleneck: a falsely shared cache line that prevents an efficient use of the cache of the processor, a critical section that reduces the parallelism, a saturated hard drive, an inefficient memory placement on a NUMA architecture that saturates an interconnect link, a barrier that does not scale with the number of cores etc.

Today, we use profiling tools to identify these bottlenecks. These tools indicate if a hardware component saturates or if a synchronization hampers the progress of a thread. These tools give raw metrics, but they do not indicate if optimizing a block of code can lead to better performance. Typically, these profiling tools report metrics to know if a memory controller [1], [2], [3], [4], [5], a cache [6], [7], [8], [9], [10], [11], a lock [12], [13], [14], [15] or the network stack [16], [17] is inefficiently used. However, these metrics indicate a saturation, but a saturation does not necessarily lead to a large slowdown, either because the access to the saturated component remains efficient or simply because the slowdown caused by the saturation has only a marginal impact on the overall performance.

Instead of just reporting raw metrics, some research works use "differential execution" to identify the possible causes of a performance bottleneck [18], [19], [20]. The intuition behind differential execution is relatively simple. If the execution time of a function (or of a request) executed many times becomes particularly slow, we just have to compare the configuration or the code executed in the slow and fast cases in order to identify the cause of the slowdown. Differential execution is an important technique to identify the causes of an abnormal execution time. However, differential execution was not designed to identify *if* an execution time is abnormal. In their works, the authors use the average execution time as a reference to identify an abnormal execution time, and suppose thus that the average execution time is good. This is, however, often not necessarily the case when a hardware component or a synchronization primitive saturates. Indeed, as presented in Section 4.4, in case of saturation, almost all the executions of a function are abnormally long, which hides the problem.

In this work, we propose to reuse the ideas behind differential execution, but we propose to use differential execution to identify the blocks of code that hamper the parallelism. For that purpose, instead of focusing on the *average* execution time, which can hide performance bottleneck, we propose to focus on the *fastest* one. Our intuition is that the fastest execution of a block of code gives a theoretical better execution. Any longer execution is probably caused by the interference from another thread when it accesses the same hardware resource or the same synchronization primitive. By using this theoretical lower bound, we can compute the theoretical time that a block of code would have taken if all the executions had been as fast as the best one. Based on this idea, we define, for a given block of code, its `SCI` (Slowdown Caused by Interference) score, which gives the theoretical slowdown caused by interference. The SCI score comes thus as a complement to the existing techniques. While differential executions and classical profiling tools can identify why a block of code suffers interference, the SCI score identifies which blocks of code suffer interference, if the interference leads to a large slowdown, and thus if optimizing the blocks of code would lead to better performance.

In order to evaluate the usability of the SCI score, we developed a profiling toolchain, called ISPOT (*Interference Spotter*), that computes the SCI scores of a set of functions provided by the user. We use ISPOT on 27 applications from 6 benchmarks to evaluate if the `SCI` score can actually identify parallelism bottlenecks. We found that the `SCI` score is able to identify bottlenecks caused by false sharing, contended locks, saturated networks, saturated hard drives, imbalanced workloads and inefficient NUMA placements, with few and easily identifiable false positives. In detail, we found that:

- ISPOT detects interference in 14 time consuming functions from 10 of the evaluated applications.
- Among the 14 functions, 2 (13%) are false positives that appear when the variation of a function is caused by a varying workload. We show that these false positives are easy to manually identify and to discard.
- The remaining 12 functions pinpoint actual interference problems caused by false sharing, lock contention, imbalanced load, NUMA memory placement, network stack and disk I/O. 7 interference bottlenecks were previously identified in other works, while 5 are new.
- Based on this analysis, we can correct 8 functions by modifying at most only 25 lines of code, which leads to a performance improvement of up to 9 times.

The paper is organized as follows. Section 2 presents the SCI metric, while Section 3 presents ISPOT. We then evaluate ISPOT with micro-benchmarks in Section 4 and with complete applications in Section 5. Finally, Section 6 presents the related work and Section 7 concludes the paper.

## 2 The SCI metric

The SCI metric aims at identifying the effect of interference on a multi-threaded application. This metric identifies (i) where the code suffers from interference and (ii) how much the interference impacts the performance.

In order to define the SCI metric, we start with a simple observation: interference slows the execution down. However, capturing the slowdown caused by interference is difficult because we cannot easily know the execution time of a code in absence of interference. We could try to run a thread in isolation, but it is not always possible because threads often synchronize and interact. We could also vary the number of threads or even the workload, as proposed by OSProf or statistical debugging tools [18], [21], [22]. This solution is not satisfying because changing the setting often drastically changes the executed code. For example, with OpenMP, varying the number of threads of a parallel loop changes the number of iterations executed by each thread, which makes the comparison between the runs difficult.

In order to compute the slowdown caused by interference, we rely on statistics: when we execute often a block of code, it will sometimes execute almost without interference. We thus consider that the fastest execution is almost interference-free, and that any slower execution is caused by interference. This interference can be caused by a contention on a resource shared with another thread or process, or by another problem (eg. kernel scheduling). We can then compute, for any block of code, the slowdown of a thread caused by interference. We define this metric as the SCI score (Slowdown Caused by Interference) of a block of code. Formally, the SCI of a block of code is the sum of all $(d_j - \overline{d_i})$ (where $\overline{d_i}$ is the duration of the fastest execution of this block, $d_j$ is the $j^{th}$ execution occurrence of the block of code) divided by the thread duration.

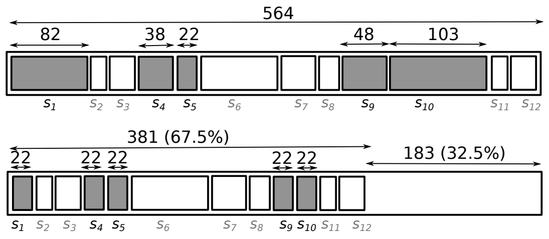

564
82
38
22
48
103
$s_1$ $s_2$ $s_3$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$
381 (67.5%)
183 (32.5%)
22
22 22
22 22
$s_1$ $s_2$ $s_3$ $s_4$ $s_5$ $s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$ $s_{11}$ $s_{12}$

Fig. 1: Illustration of the SCI metric: based on the original execution trace (on the top), an ideal improvement without interference is estimated at 32.5% (on the bottom)

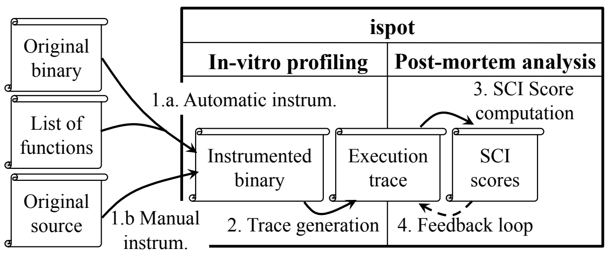


Fig. 2: Overview of ISPOT.

Figure 1 illustrates the principle. Each box gives the time taken by the execution of the blocks of code of an application, and the gray boxes gives the execution of a unique block of code. If we assume that the execution with the minimum duration (here, 22) is only slightly slowed down by interference, the SCI score approximates the performance improvement (represented at the bottom) that would have been obtained if each occurrence of the block of code represented in gray had been executed without interference.

### 2.1 Validity

In order to define the SCI metric, we consider that performance variation mostly comes from interference. In practice, this is not the case because of low-level hardware mechanisms. Typically, we ignore the warm-ups of the caches, of the branch predictors or of the cache-line prefetchers. However, this performance variation often affects only the first occurrences. As we record thousands to millions of occurrences, these first occurrences only marginally modify the SCI score. Moreover, we consider that the fastest execution of a block of code is almost interference-free. In practice, we cannot prove that this fastest execution does not suffer from interference. However, since we record thousands to millions of occurrences, the probability of executing a block of code almost without interference should be large. We confirm this hypothesis in Section 4 by experimentally showing that the fastest execution of a block of code takes always almost the same time when we increase the contention in four different micro-benchmarks, even in very highly contended cases.

## 3 Design and implementation of Ispot

ISPOT (Interference Spotter) automatically computes the SCI scores of blocks of code by running the application in-vitro (see Figure 2). During this in-vitro run, ISPOT records a timestamp when a thread enters or leaves a block of code.

Based on these timestamps, ISPOT performs a post-mortem analysis to compute the times taken by each execution of a block of code, which are then used to compute the SCI score.

As illustrated in Figure 2, we have chosen to consider two use cases. In the first case (1.a. Automatic instrumentation), the user gives the binary and a set of function names to ISPOT, which automatically instruments the functions of the set.[1] In the second case (1.b. Manual instrumentation), the user manually instruments the source code because the user wants to analyze a block of code that is finer than a function. In this case, the user manually inserts two lines of code in the source file to profile a block of code: one at the beginning of the block, the other at the end. After the instrumentation, ISPOT runs the application, records the timestamps and computes the SCI score. We explain these steps in detail in the remainder of the section.

## 3.1 Automatic instrumentation of the application

When the user provides a list of functions, ISPOT automatically instruments the application (step 1.a of Figure 2). For that purpose, ISPOT relies on the EZTRACE tracing framework [23]. In detail, ISPOT extracts the function prototypes from the debugging symbols (Dwarf tables), and then uses EZTRACE to instrument the binary of the application.

EZTRACE can use two different methods for instrumenting a function. If the function is located in a shared library, EZTRACE uses `LD_PRELOAD` to intercept the calls to the function. If the function is located in the binary, EZTRACE patches the binary when the application is loaded in memory by the ELF loader [24]. In both cases, EZTRACE replaces a call to the original function by a wrapper provided by ISPOT. The wrapper first records an event (marking the beginning of the function), then calls the original function, and finally records another event (marking the end of the function).

## 3.2 Trace generation

As presented in Figure 2 (2. Trace generation), ISPOT runs the instrumented version of the application and records the events used to compute the SCI scores in an execution trace. An event consists of a timestamp (the CPU cycle counter), the function name, a marker to know if the thread enters or leaves the function, and the function arguments.[2] We detail how ISPOT records events in the remainder of this section.

### *3.2.1 Recording the events*

ISPOT often records millions of events. For instance, the `DC` application generates a 17 GiB trace that contains 364 millions events. Since recording this trace in a file can drastically changes the timing behavior of the application, ISPOT batches the I/Os. To avoid thread synchronizations, each thread stores its events in its own pre-allocated buffer. When a buffer is full, ISPOT flushes the buffer to disk and reinitializes the buffer. We ensure that the buffers are rarely flushed during the run by using large buffers. On our small machine (Xeon4, see Section 4), we pre-allocate 1 GiB for the buffers (1/8 of the memory), while on our large machine (Opteron48, see Section 4), we pre-allocate 32 GiB (also 1/8 of the memory).

### *3.2.2 Recording the call stacks*

In addition to the recorded events, ISPOT records call stacks to simplify the analysis. For example, with the `pthread_mutex_lock` function, the developer wants to know which critical section is protected by the lock acquisition in order to optimize the code. Systematically recording the call stack when a thread executes an instrumented function would lead to a large slowdown. For this reason, ISPOT records a call stack every N invocations, beginning with the first invocation.

In the 27 applications tested in our experiments, all the instrumented functions have few call sites, and each call site only has to be captured once in order to help the user to identify the source of a problem. In all our experiments, we configured thus N to 10 000 because (i) this period is large enough to prevent a slowdown caused by stack recording and (ii) this period is small enough to allow ISPOT to capture all the call sites of an instrumented function.

### *3.2.3 Mitigating the instrumentation overhead*

As a result of our optimizations, the instrumentation overhead of a function is most of the time reduced to a function call (from the wrapper to the original function), two accesses to the timestamp counter, the copy of few bytes in the buffer, and, only when EZTRACE modifies a binary, two `jmp` instructions. On the machines used for the evaluation (see Section 4), we have measured that the overhead of an instrumented function always remains below 100 ns, while recording two timestamps already takes 80 ns on our machines.

However, even if the overhead to record a single timestamp is low, this overhead can become large if the instrumented function is short and called often. In our experiments, we have observed a worst case of 30%, which can change the timing behavior of the application and make the SCI score inaccurate. When the overhead becomes too large, the user can use a sampling mode: ISPOT only records a timestamp every N function invocations, where N is specified by the user. This mode trades the overhead for a lower accuracy because the sampling mode reduces the probability of capturing an interference-free execution of a function.

## 3.3 SCI score computation

To compute the SCI scores from an execution trace, ISPOT first identifies the repetitive sequences of events in the trace [25]. ISPOT identifies the repetitive sequences with more than two events in order to handle nested calls: typically when a function `f` calls a function `g`, a call to `f` is represented by the `f_start g_start g_end f_end` sequence of events in the trace. In order to illustrate the algorithm, the following example shows a group of events that appear in this order in the trace of one thread:

$$\underline{\mathbf{a}\ \mathbf{b}\ \mathrm{c}}\ \mathrm{b}\ \mathrm{d}\ \mathbf{a}\ \mathbf{b}\ \mathrm{e}\ \underline{\mathbf{a}\ \mathbf{b}\ \mathrm{c}}$$

The algorithm aims at identifying that the sequence of events `a b` (in bold) is a repetitive sequence that appears

1. The function name is the mangled name for C++ applications.
2. In case of a manual instrumentation, the user can replace the function arguments by custom arguments.

3 times, and that the sequence `a b c` (underlined) is a repetitive sequence that appears twice. To summarize, the algorithm starts with the first pair of events (`a b` in our case). It tries to find this sequence in the trace. If it finds another occurrence, the algorithm identifies a repetitive sequence `a b` and replaces all its occurrences in the trace by a meta-event that contains `a` and `b`. In our example, the algorithm continues with the new pair `(meta a b) c`. It applies the same algorithm, and thus replaces all the occurrences of `(meta a b) c` by a new meta-event. The algorithm continues with the pair `(meta a b c) b`. As it does not find any occurrence of this pair, the algorithm continues with `b d`, `d (meta a b)`, `(meta a b) e` and `e (meta a b c)` before terminating.

ISPOT then computes the SCI score of all the repetitive sequences of events and reports the name of the event (i.e. the function name), the SCI score, the number of occurrences of the sequence, and the recorded call stacks.

### 3.4 False positives

ISPOT assumes that, for a given block of code, its execution time is slower than another one because the longest execution is slowed down by interference. While this assumption holds in many cases, it may also lead to *false positives*. Some executions take more time because of the workload: because of the function arguments or because of the global state (global variables, operating system state). This is typically the case of a function that searches an element in a linked list.

In order to prevent false positives and thus to accurately identify the repetitive sequences of events, ISPOT should ideally automatically identify which arguments or which global variables change the workload of a block of code. However, identifying these arguments and variables in general is difficult. First, static analysis does not help because static analysis cannot identify the actual control flow taken during the execution. Typically, a static analysis will report that an error handling code never executed can lead to different execution times. With static analysis, for the functions with an error handling code (which are numerous), the arguments of the functions have thus to exactly matches in order to consider that they lead to the same repetitive sequence of events. This behavior drastically decreases the number of samples per repetitive sequences of events, which in turn drastically reduces the probability to capture an almost interference-free execution. Moreover, automatically identifying false positives is also difficult because we have observed that a varying workload can lead to approximately the same execution time and should be ignored in order to compute an accurate SCI score. This is the case in the DC application, where a function writes small buffers to a disk (see Section 5.5.6). The function takes approximately the same time, even when the buffer size changes, because the function spends most of its time in the system call

Because automatically identifying the false positives in general is difficult, we consider in this work that reporting false positive remains the best strategy. In our experiment, we show that, among the functions with a high SCI score, ISPOT only reports 13% of false positive. If ISPOT reports a false positive, the developer can specify which parameters matter before restarting the computation of the SCI scores (4. feedback loop in Figure 2). Moreover, ISPOT uses a default list of well-known parameter-dependent functions, such as `pthread_mutex_lock`, for which it is obvious that the workload will vary a lot with the arguments.

```
for(i=0; i<NITER; i++) {
  compute(delay);
  lock(&l);
  value++;
  unlock(&l);
}
```

Listing 1: Code of the lock contention micro-benchmark

```
struct { int x; int y; } data;

for(i=0; i<NITER; i++) {
  if(tid == 0) {
    data.x++;
  } else {
    data.y++;
    compute(delay);
  }
}
```

Listing 2: Code of the false sharing micro-benchmark

```
fd[tid]=open(file[tid], O_RDONLY|O_DIRECT);
for(i=0; i<NITER; i++) {
  compute(delay);
  read(fd[tid], buffer, block_size));
}
```

Listing 3: Code of the I/O contention micro-benchmark

## 4 MICRO-BENCHMARK EVALUATIONS

In this section, we study the SCI scores of several simple micro-benchmarks that implement known interference problems. This study has the goal of (i) verifying that the SCI score is actually correlated to an interference problem, (ii) verifying that, even when the interference problem is very frequent, ISPOT is able to capture almost interference-free executions, (iii) showing that, when the interference problem is frequent, most of the execution times are large and thus that a comparison to the average execution time is not a good indicator of a contention problem, (iv) analyzing how the sampling mode of ISPOT impacts its accuracy.

For our evaluations, we use two machines: (i) Xeon4 has 4 cores, 8 GiB of memory, 1 Intel Xeon E5-2603 socket, 1 NUMA node. Linux version: 4.4.0-1, gcc version 5.3.1, glibc version: 2.22-4, and (ii) Opteron48 has 48 cores, 256 GiB of memory, 4 AMD Opteron 6172 Dodeca-core sockets, 8 NUMA nodes. Linux version: 4.9.0, gcc version: 4.9.2, glibc version: 2.21.

### 4.1 Summary of the micro-benchmarks

We consider four different micro-benchmarks. The first and the second micro-benchmarks exhibit a problem that occurs

| Benchmark | X | $\rho$ | # samples |
|---|---|---|---|
| POSIX lock contention | lock acq. time | 0.99 | 18 |
| spinlock contention | lock acq. time | 0.95 | 16 |
| false sharing | time to access x | 0.95 | 12 |
| io contention | read time | 0.99 | 11 |

TABLE 1: Correlation coefficient for the micro-benchmarks.

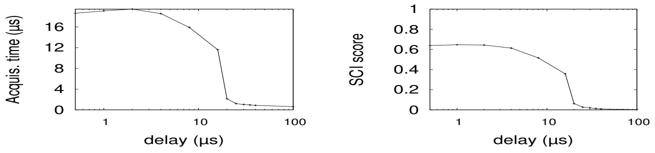
when multiple threads try to acquire the same lock at the same time. In case of contention, the cache line that holds the lock variable continuously bounces between the cores, which saturates the buses between the cores (e.g., the interconnect on Opteron48). This saturation drastically increases the time to acquire a lock and leads to a performance collapse [26]. Listing 1 reports the code used to evaluate lock contention. We use a POSIX lock in the first micro-benchmark and a spinlock in the second. At each iteration, a thread simulates a computation for `delay` µs, acquires a lock, increments a shared variable, and then releases the lock. We execute this micro-benchmark on Opteron48 with 47 threads (in order to let an idle core so that the OS can schedule other ready processes), and we vary `delay` to simulate different levels of lock contention.

The third micro-benchmark suffers from false sharing. False sharing appears when multiple threads access different variables that happen to be located on the same cache line [27], [28]. Each thread, by updating its own variable, invalidates the cache line for the other threads, which leads to cache misses and performance degradation. This problem is hard to detect because source code analysis does not show any explicit relationship between the variables located on the same cache line. Listing 2 reports the code used to evaluate false sharing. The first thread (`tid = 0`) continuously updates its variable `x`. The other thread (`tid = 1`) updates its independent `y` variable and then simulates a computation by executing `delay` iterations of an empty loop. As `x` and `y` are located on the same cache line, the access of the second thread invalidates the cache line for the first thread. We execute this benchmark on Xeon4 with 2 threads and we vary the delay to simulate different probabilities of false sharing.

In the fourth micro-benchmark, many threads perform I/O operations simultaneously. Since the disk may saturate, we have a typical case of interference, which may lead to a large overhead. Listing 3 reports the code used to evaluate I/O contention. Each thread opens its own file and reads it sequentially with a delay between read operations. In order to bypass the I/O cache of the operating system, each thread opens its file with the `O_DIRECT` flag, which ensures that every call to `read` actually triggers a physical I/O. We run the micro-benchmark on Opteron48 with 47 threads, and each thread reads blocks of 512 bytes, while varying delay from 0 to 4 ms in order to evaluate different levels of I/O contention.

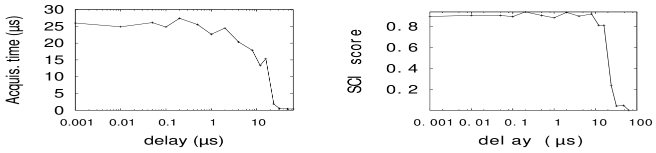
### 4.2 Correlation between the SCI score and an interference

Figures 3, 4, 5 and 6 report the evaluation of the micro-benchmarks. In each figure, (a) gives the average performance when we vary `delay`, and (b) gives the SCI score when we vary `delay`. For the two lock micro-benchmarks,



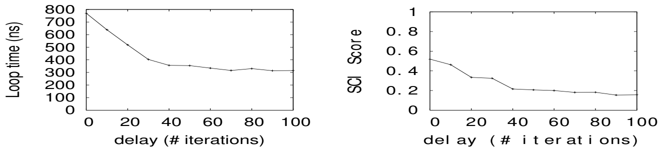
(a) Acquisition time (b) SCI score

Fig. 3: Lock contention micro-benchmark (POSIX lock)



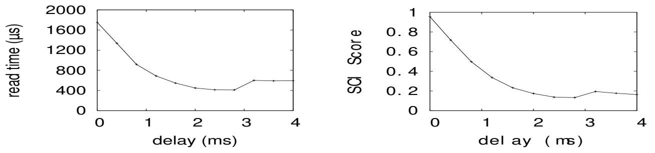
(a) Acquisition time (b) SCI score

Fig. 4: Lock contention micro-benchmark (spinlock)



(a) Read time (b) SCI score

Fig. 5: False-sharing micro-benchmark



(a) Read time (b) SCI score

Fig. 6: I/O contention micro-benchmark

(a) reports the completion time to acquire a lock. For the false sharing micro-benchmark, (a) reports the completion time to read the variable. Finally, for the I/O contention micro-benchmark, (a) reports the completion time of a read operation.

As expected, for all the micro-benchmarks, by observing the (a) figures, we can see that when the delay increases, the completion time decreases since the probability of interference decreases. Moreover, we can observe in the (b) figures that the SCI score seems to behave exactly as the completion time: when the delay increases, the SCI scores also decreases.

In order to confirm this observation, we compute the Pearson product-moment correlation coefficient between the completion time and the SCI score for each micro-benchmark. The correlation coefficient $\rho(X, Y)$ of two random variables $X$ and $Y$ is a number between -1 and 1. A value close to -1 or 1 indicates that a linear relation exists between the two variables. Table 1 reports the correlation coefficient between the completion time and the SCI score for each micro-benchmark, along with the number of samples (points on the x-axis) for each variable. We can observe that the coefficient is high in all the experiments (above 0.95), which confirms the linear relation between the completion time and the SCI score in the micro-benchmarks. From this

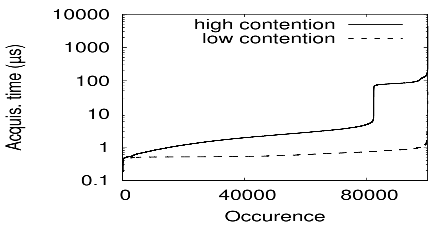


Fig. 7: POSIX lock acquisition times ordered by execution time (`delay`=100µs for the low contention and 0µs for the high one).

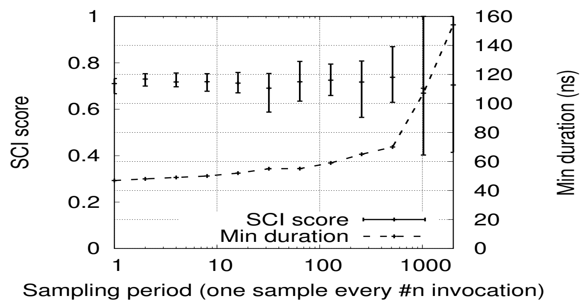


Fig. 8: Evolution of the SCI score as the sampling period changes

strong correlation, we can conclude that the SCI score captures the performance degradation caused by interference in the micro-benchmarks.

### 4.3 Impact of the contention on the fastest execution

As presented in the introduction, we suppose that, when we execute often a block of code, the probability to capture an almost interference-free execution is large. In order to validate this hypothesis, Figure 7 reports the execution times of the critical sections of the POSIX lock micro-benchmark ordered by the execution time. We can observe that the fastest occurrences (on the left on the figure) takes approximately the same time whatever the contention. We do not report the same figures for the other micro-benchmarks because of a lack of space, but they also show that the fastest executions take approximately the same time whatever the contention. These results confirm our hypothesis: the probability to capture an almost interference-free execution is large when we execute often a block of code, even when the contention is high.

### 4.4 Impact of the contention on the distribution

Differential execution analysis consists in comparing the duration of a piece of code to a reference execution. In several research works [18], [19], [20], the reference execution is either manually pinpointed by a human, or automatically selected using the average duration of the piece of code. As shown in Figure 7 (which reports the execution times in the POSIX lock micro-benchmark, see Section 4.3), most of the executions are slowed down by interference in case of high contention. This result confirms that considering that all the executions slower than the fastest one as abnormal gives a better indication of a contention problem than only considering the executions slower than the average execution time.

### 4.5 Evaluation of the sampling period

As described in Section 3.2.3, the overhead of ISPOT can be mitigated using a sampling mode that consists in recording a timestamp every N function calls. In order to evaluate the sensitivity of the SCI score when the sampling mode is activated, we run the POSIX micro benchmark while varying the sampling period. We set `delay` to 1 µs (high contention) and run the experiment 10 times on Opteron48. Figure 8 reports the SCI score and the minimum duration detected as the sampling period changes. The SCI score error bars show the minimum, average, and maximum SCI scores observed.

The results show that when the sampling period is low, the SCI score remains stable from one run to another. When ISPOT records timestamp less often, the SCI score may vary from one run to another, but still indicates an interference problem: for instance, when recording a timestamp every 512 function invocations, the SCI score varies from 0.63 to 0.87. When the sampling period becomes high, the SCI score varies considerably because of the small number of samples.

The figure also shows that the minimum duration (see the dotted line) captured by ISPOT slightly increases as the sampling period decreases. This validates once again the hypothesis formulated in Section 2: when the number of occurrences of a repetitive sequence is large, the probability of capturing an almost interference-free occurrence is large.

## 5 APPLICATIONS EVALUATION

In this section, we evaluate ISPOT with real applications. This evaluation has first the goal of verifying that ISPOT can actually identify the effects of interference in real applications. For that purpose, we use ISPOT to evaluate a panel of 27 applications. This evaluation also has the goal of showing that, by cross-checking the interference bottlenecks identified by ISPOT with the reports of other profiling tools, we can often easily optimize the applications.

After a presentation of the applications, this section describes how we generate the list of functions instrumented by ISPOT. The section then presents an analysis of the possible false negatives when we instrument these functions, a systematic analysis of the functions with a high SCI score and an analysis of the false positives reported by ISPOT.

### 5.1 Evaluated applications

We evaluate 27 applications summarized in Table 2. We have selected these applications because they are widely used to evaluate the parallelism and because some of them are already known to suffer from interference [26], [28], [29], [30], [31].

Phoenix-2 [32] is a MapReduce for shared-memory systems written in C. It comes with small sample applications with data sets ranging from 59 to 512 MiB. The Splash2 benchmark [33] contains small multi-threaded C applications, ranging from a ray tracer to a large-scale ocean movement computation. The Parsec 2.1 benchmark [34] contains moderate to large multi-threaded C++ applications from various fields such as financial analysis or data-mining.

| | Pheonix-2 | Splash2 | Parsec | NPB (C) | Memcached | LevelDB | Total |
|---|---|---|---|---|---|---|---|
| Hardware setting | Xeon4 | Opteron48 | Opteron48 | Opteron48 | 2*Xeon4 | Xeon4 | |
| # threads | 4 | 48 | 48 | 48 | 1*4 + 4*1 | 4 | |
| # applications | 7 | 7 | 4 | 7 | 1 | 1 | 27 |
| # manually instrumented | 7 | 0 | 0 | 0 | 1 | 0 | 8 |
| # applications with a high score | 2 | 2 | 1 | 3 | 1 | 1 | 10 |
| # functions with a high score | 2 | 3 | 1 | 6 | 2*1/2 | 1 | 14 |
| # false positives | 1 | 0 | 0 | 1 | 0 | 0 | 2 |
| # applications with true defects | 1 | 2 | 1 | 3 | 1 | 1 | 9 |
| # functions with true defects | 1 | 3 | 1 | 5 | 1 | 1 | 12 |
| Discussed in Section | 5.5.2 | 5.5.1, 5.5.1, 5.5.2 | 5.5.4 | 5.5.4, 5.5.5, 5.5.6 | 5.5.3 | 5.5.4 | |
| # functions never reported | 0 | 1 | 0 | 4 | 0 | 0 | 5 |
| # corrected functions | 1 | 3 | 0 | 5 | 1 | 0 | 8 |

TABLE 2: Summary of the evaluated applications

NPB (NAS Parallel Benchmark 3.3 [35]) contains moderate to large Fortran and C applications, ranging from linear algebra to a data mining application, which writes 2.5 GiB of data in the file system [36]. We use the OpenMP implementation of NPB with the large class C dataset. Memcached 1.4.36 [37] is an in-memory cache widely used for web servers. We evaluate memcached with the memaslap client [38], which generates 70% of set and 30% of get during 30s. We run memcached with 4 threads in multi-threaded mode on a Xeon4 machine and 4 mono-threaded instances of memaslap on another Xeon4 machine. LevelDB 1.20 [39] is a fast key-value store library, shipped with the `db_bench` benchmark. In our setting, each of the four threads inserts one million random values in the database. In all our experiments and for each configuration, we run the application at least 5 times and we did not observe significant variations.

### 5.2 Identification of the hottest functions

As presented in Section 3, ISPOT automatically computes the SCI scores of a list of functions provided by the developer. In our experiments, we analyze the most time consuming functions as reported by `Linux perf`. We have chosen to analyze the functions that take more than 1% of the total execution time, because we consider that improving the performance by removing interference from a colder function is unlikely. Table 3 reports, for each application, the functions that take more than 1% of the total execution time. In total, we analyze 70 functions from 27 applications with ISPOT.

### 5.3 Analysis of the potentially false negatives

As presented in Section 3, if a function is called few times, the probability of capturing an interference-free execution becomes low. In this case, the SCI score is low because ISPOT cannot identify a potential interference bottleneck. Among the 70 analyzed functions, 6 are in this case. They all come from the Phoenix-2 benchmark (the 6 first functions in Table 3). These functions are called once. The SCI score is thus systematically equal to 0, which may hide an interference bottleneck.

For these 6 potentially false negatives, the instrumented functions contain a large loop executed many times. We have manually instrumented these functions in order to record a timestamp every ten iterations of the loop, because recording more timestamps leads to a larger slowdown.

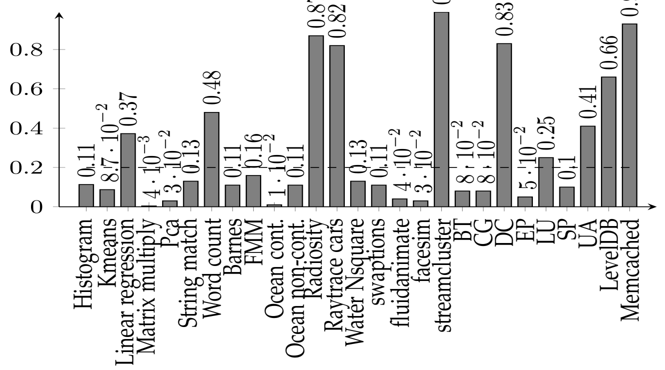


Fig. 9: Highest SCI scores

### 5.4 Instrumentation overhead

The last column of Table 3 reports the overhead caused by instrumentation without activating the sampling mode of ISPOT (manual instrumentation for the 6 potentially false negatives and automatic instrumentation instead). This overhead is computed by comparing the application execution time with and without instrumentation. Since the computation of the SCI score is performed offline, it is excluded from the overhead computation.

Our evaluation shows that the overhead remains below 10% for 22 applications. For 5 applications, however, we observe an overhead above 10% with a maximum of 31%. As a comparison, the mean overhead reported in the Coz paper is 17%, and the maximum measured overhead is 65% [40]. The mean overhead of ISPOT is equal to 8% and its maximum to 31%. We consider thus that the overhead of ISPOT remains reasonable for an in-vitro profiling, and that this overhead should not drastically change the behavior of the applications. Moreover, the developer can always trade this overhead for a reduced accuracy by activating the sampling mode of ISPOT (see Section 3.2.3).

### 5.5 Analysis of the SCI scores

Figure 9 reports the highest SCI scores identified by ISPOT. For the analysis, we focus on applications with an SCI score greater than 0.2. With this threshold, a thread loses more than 20% of its time because of interference. Thus, we consider that it becomes interesting to understand why. We identify 14 functions from 9 applications with a SCI score higher than 20%. The remainder of the section presents

|  | Application | Function | % of time | Instrum. overhead |
|---|---|---|---|---|
| Phoenix-2 | histogram | calc_hist | 94% | 6.42% |
|  | kmeans | find_clusters | 79% | 10.53% |
|  |  | calc_means | 21% |  |
|  | linear_regression | linear_regression_pthread | 99% | 8.7% |
|  | matrix_multiply | matrixmult_map | 99% | 0.12% |
|  | pca | calc_cov | 99% | 1.23% |
|  | string_match | string_match_map | 88% | 2.94% |
|  | word_count | __strcmp_sse2_unaligned | 44% | 28.99% |
|  |  | wordcount_reduce | 27% |  |
|  |  | __memmove_ssse3_back | 22% |  |
| Splash-2 | Barnes | hackcofm | 65% | 23.18% |
|  |  | walksub | 12% |  |
|  |  | subdivp | 12% |  |
|  | FMM | VListInteraction | 33% | 8.47% |
|  |  | DownwardPass | 32% |  |
|  |  | UpwardPass | 24% |  |
|  | Ocean cont. | pthread_barrier_wait | 23% | 5.96% |
|  |  | relax | 14% |  |
|  |  | slave2 | 12% |  |
|  |  | __lll_lock_wait | 12% |  |
|  |  | __lll_unlock_wake | 11% |  |
|  | Ocean non cont. | slave2 | 21% | 7.39% |
|  |  | laplacalc | 16% |  |
|  |  | relax | 12% |  |
|  |  | pthread_barrier_wait | 11% |  |
|  | Radiosity | __lll_unlock_wake | 37% | 27.58% |
|  |  | __lll_lock_wait | 24% |  |
|  |  | _process_task_wait_loop | 21% |  |
|  | Raytrace car | __lll_unlock_wake | 52% | 9.05% |
|  |  | __lll_lock_wait | 31% |  |
|  | Water-nsquared | _int_malloc | 40% | 5.08% |
|  |  | INTERF | 17% |  |
|  |  | CSHIFT | 16% |  |
| Parsec-2.1 | swaptions | HJM_SimPath_Forward... | 33% | 1% |
|  | fluidanimate | ComputeForcesMT | 48% | 5.8% |
|  |  | ComputeDensitiesMT | 32% |  |
|  | facesim | Add_Force_Differential | 29% | 6.5% |
|  |  | Update_Position_Based... | 17% |  |
|  | streamcluster | T.203 | 44% | 5% |
|  |  | parsec_barrier_wait | 41% |  |
| NPB 3.3 | BT | compute_rhs_ | 30% | 5% |
|  |  | x_solve_ | 10% |  |
|  |  | y_solve_ | 13% |  |
|  |  | z_solve_ | 13% |  |
|  | CG | conj_grad_ | 73% | 2% |
|  | DC | KeyComp | 26% | 5.74% |
|  |  | MultiWayMerge | 24% |  |
|  |  | __memcpy_sse2 | 19% |  |
|  |  | __write_nocancel | 13% |  |
|  | EP | __ieee754_log_sse2 | 48% | 1% |
|  |  | vranlc_ | 22% |  |
|  | LU | sync_left_ | 35% | 0.05% |
|  |  | rhs_ | 22% |  |
|  |  | sync_right_ | 14% |  |
|  |  | jacu_ | 6% |  |
|  |  | buts_ | 6% |  |
|  |  | ssor_ | 5% |  |
|  |  | jacld_ | 5% |  |
|  |  | blts_ | 5% |  |
|  | SP | compute_rhs_ | 36% | 3% |
|  |  | z_solve_ | 17% |  |
|  |  | x_solve_ | 16% |  |
|  |  | y_solve_ | 15% |  |
|  | UA | gomp_team_barrier_wait_end | 24.01% | 30.86% |
|  |  | gomp_barrier_wait_end | 23.58% |  |
| LevelDB |  | __GI___libc_write | 21% | 6.9% |
|  |  | pthread_cond_signal | 10% |  |
|  |  | pthread_cond_wait | 9% |  |
| Memcached |  | sendmsg | 36% | 0% |
|  |  | epoll_wait | 24% |  |

TABLE 3: Time consuming functions reported by `Linux perf`

```
pthread_mutex_lock(&(gm->ridlock));
ray->id = gm->rid++;
pthread_mutex_unlock(&(gm->ridlock));
```

Listing 4: code of the hottest function in Raytrace

an exhaustive analysis of these functions. The section also presents optimizations that solve the bottlenecks in eight of the functions (use of another lock algorithm in Section 5.5.1, false sharing mitigation techniques in Section 5.5.2, better deployments in Sections 5.5.3 and Section 5.5.5).

### 5.5.1 Lock contention

ISPOT reports high SCI scores for the lock acquisition function in two applications: Raytrace (SCI of 0.82) and Radiosity (SCI of 0.87). Thanks to the call stacks reported by ISPOT, we can easily identify where is the interference bottleneck in the code.

Listing 4 presents the code associated with the lock acquisition in Raytrace. The high SCI score is due to the high contention on this lock. Lozi et al. [26], [29] also identify this saturation, and we have therefore reused their algorithm (the RCL lock). We confirm their result: using an RCL lock divides by 4 the completion time of Raytrace and by 5.37 the completion time of Radiosity. This experiment confirms that the SCI metric is able to identify contended locks. With the modified applications, we observe a high score of 0.25 in Raytrace and a low score of 0.11 in Radiosity. False sharing causes the high score of Raytrace and we discuss this case in the next section.

### 5.5.2 False sharing

ISPOT identifies two functions that suffer from false sharing, one from Raytrace that was never reported (SCI of 0.25), the other from Linear_regression that was previously reported by Liu et al. [28] (SCI of 0.37).

The high SCI score in Raytrace appears after correcting the lock contention (see Section 5.5.1), and in the same function (see Listing 4). In order to understand the cause of the interference, we use `oprofile` to compute the number of cache misses per function. We observe a high number of cache misses in the code reported in Listing 4. As false sharing often causes an unexpectedly high number of cache misses, we have simply added padding around `gm->rid`. Thanks to this modification, we improve the performance of Raytrace by 15%, which, with the lock optimization (see Section 5.5.1), leads to a completion time divided by 4.81.

In Linear_regression, measuring the cache misses with `oprofile` highlights a large loop that accumulates its result in a structure falsely shared with the other threads. In order to solve the problem, we accumulate the results in local variables and only propagate the result in the falsely-shared structure at the end of the loop. We improve the performance of Linear_regression and divide the completion time by 8.87.

Thanks to these optimizations, we have eliminated the high SCI scores in both applications.

| Network configuration | SCI score | | Transaction/s | Network rate |
|---|---|---|---|---|
| | client | server | | |
| 100 Mib | 0.93 | $< 0.1$ | 36k | 11 M/s |
| 1 Gib | 0.82 | $< 0.1$ | 193k | 61 M/s |
| Local loop | 0.62 | $< 0.1$ | 2 037k | 632 M/s |

TABLE 4: Correlation between the network saturation and the SCI score in memcached.

### 5.5.3 Network contention

When ISPOT automatically instruments the time consuming functions of memcached, ISPOT reports very low SCI scores. This result shows that, in our experiment, memcached does not seem to suffer from interference. However, by only instrumenting the server, we cannot identify an interference bottleneck on the network between the client and the server. For this reason, we have manually instrumented memcached (the server) and memaslap (the client) in order to compute the completion time of a request from both the client and the server sides. Table 4 presents the result for different network configurations: a network at 100 Mib, a network at 1 Gib and a local network loop when we co-localize the client and the server on a 32-core Intel Xeon E5-1607. The low SCI score of the server with each configuration confirms that the server does not suffer from interference. Moreover, the SCI scores at the client side decrease when the network bandwidth increases. These results show that the network suffers from interference, and that the SCI score accurately identifies the contention. We can also see that the SCI score remains high when we co-localize the server and the client on the same machine, which shows that the network remains a bottleneck even in this case.

### 5.5.4 Parallelism

We have identified a problem of parallelism in LevelDB also identified by the developers of RocksDB [41]. The function `pthread_cond_wait` has a high SCI score of 0.66. After an analysis of the code, we found that the application inserts new keys in mutual exclusion during a compaction by using this variable condition function. As performing a write is slow, the writes in mutual exclusion hamper the parallelism. The SCI score is high because in some rare cases, the monitor is free, while often, a thread has to wait for the other threads before entering the monitor. We cannot fix this issue without deeply redesigning LevelDB, but the developers of RocksDB state that, by compacting the memtables in parallel, they are able to multiply by up to 10 the performance during compaction. We also confirm our observation by measuring the scalability of LevelDB: the duration of operations quickly increases when the number of threads increases (2.4 µs/op with 1 thread, 8.8 µs/op with 2 threads, 20.1 µs/op with 4 threads and 47.8 µs/op with 8 threads). This result confirms that LevelDB is unable to scale when the application executes many insert operations concurrently.

In streamcluster, the `parsec_barrier_wait` function has a high SCI score of 0.99 (already reported in [30], [31]). The execution trace shows that the 48 threads of the application synchronize repeatedly with this barrier function. Therefore, the threads of the application spend most of their time waiting for the other threads. The SCI score is high because in some rare cases a thread traverses the barrier quickly. Correcting this interference bottleneck would require a large code rewriting.

UA suffers from a similar problem: UA repeatedly executes parallel loops and synchronizes with an OpenMP barrier, which has a high SCI score (0.41). Similarly, correcting this interference bottleneck would require a large code rewriting.

### 5.5.5 NUMA memory placement

We have identified an interference bottleneck caused by NUMA memory placement in the LU application. A NUMA architecture connects a set of NUMA nodes by a network called the interconnect. Each NUMA node contains a set of cores and a memory controller. On a NUMA architecture, when many cores access memory located on different NUMA nodes, the interconnect or some memory controllers can saturate.

For LU, ISPOT generates a trace with 30.8 million events (1.3 GiB). ISPOT reports 3 functions with a high SCI score: `sync_left` (SCI of 0.25), `rhs` (SCI of 0.24), and `buts` (SCI of 0.20). In order to understand why these functions suffer from interference, we compute the number of memory accesses generated on each NUMA node. We identify a large memory imbalance on a single node: the master thread loads a large matrix, which is pinned on a single memory node, and, during the run, the slave threads access this matrix. As a result, the NUMA node of the matrix saturates, which explain the high SCI scores. We eliminate the imbalance by using an interleaved allocation policy, which spreads the memory on all the NUMA nodes. This NUMA policy eliminates both the high SCI scores and the memory imbalance. The completion time thus drops from 101s to 64s.

### 5.5.6 I/O contention

We have observed a problem of interference caused by I/O contention in the DC application that was not reported. By instrumenting the four hot functions in DC, ISPOT generates a trace with 364.7 million events (17 GiB). ISPOT reports two functions with a high SCI score: `MultiWayMerge` (SCI of 0.83) and `_write_nocancel` (0.33). `MultiWayMerge` is a false positive discussed in Section 5.6.

For `_write_nocancel`, each thread of the application calls this function 3.8 million times with a data size that varies between 1 and 24 bytes. This write function from the standard C library is obviously parameter-dependent, as its workload is proportional to the data size. However, we have observed that, in DC, the size of the data only marginally impacts the completion time of this function. Therefore, we have decided to consider this function as parameter-independent. Furthermore, the completion time of `_write_no_cancel` has a large variation caused by a phenomenon that is not related to the size of the written buffer. This suggests that the main problem in this application is a contention on the I/O stack.

Solving the problem requires a deep rewriting of the code. However, to verify that the I/O stack is the bottleneck, we use a RAMFS partition to store the output file of the application. The resulting performance is improved by 68% because the RAM has a better throughput. This result alone does not highlight the interference problem on the I/O stack. However, we confirm that the high SCI score is caused by interference on the disk I/O stack, because the maximum SCI score that we found by using a RAMFS drops from 0.83 to 0.17. We also confirm that the I/O stack saturates by measuring the I/O rate. `iostat` reports an I/O rate of 178 MiB/s for DC, which is slightly above the maximum I/O rate reported by `hdparm` (162 MiB/s for a sequential read).

## 5.6 Analysis of the false positives

Overall, we found 2 false positives caused by parameter-dependent functions. We present in detail an analysis of these functions, and show that identifying these functions as false positive is relatively easy.

### 5.6.1 Word_count

In `word_count`, ISPOT isolates a single function with a high score (`wordcount_reduce` with a score of 0.48). An analysis of the code shows that this high score is a false positive. We can quickly understand that the time variation is inherent to the algorithm rather than related to interference between threads. The algorithm first searches for a word in a sorted array of words. If the word is not found, it is inserted inside the array, which leads to many memory copies. The completion time of this function varies a lot: very fast occurrences of the function correspond to words that are quickly found (36 cycles), while long occurrences happen when the word is not found and when a large portion of the array moves (17 000 cycles on average).

### 5.6.2 DC

The `MultiWayMerge` function of DC has a high score (0.83) and is a false positive. By analyzing the source code of this large function, we found that the variation of its execution time is due to the merge algorithm that it implements, whose complexity depends on the input data. This high score is thus a false positive, and was relatively easy to identify in 2 hours, while we were discovering the code. We suppose that the developers of the application would have also quickly discarded this function.

## 5.7 Comparison with Coz

In order to assess the usability of the SCI score for spotting interference problems in applications, we compare ISPOT with COZ [40], a state of the art causal profiler, when running the LevelDB application.

To analyze the application with COZ, we manually insert COZ instructions at the application progress point. COZ then automatically samples the executed code and slows down the other parts of the code in order to estimate a relative speedup that the developer can expect when the not-slowed down part is optimized. As a result, COZ outputs a list of functions along with an estimation of their impact on performance. COZ selects 8 subfunctions from LevelDB, and estimates how they impact the overall performance. 3 of the functions are expected to impact the performance: `InternalKeyComparator::Compare` that compares two keys, `VersionSet::SetLastSequence` that affects a 64 bits value, and `EncodeVarint32` that encodes a 32-bit int into a variable length int. These 3 functions consist of few lines and an analysis with profiling tools shows no easy optimization: these functions do not seem to saturate a hardware component or a synchronization primitive. We have also inlined these functions and have not observed any improvement. As a result, while optimizing these functions can probably lead to an improvement, we were unable to exploit the information reported by COZ to optimize the code.

Using our approach, ISPOT automatically instruments the most time consuming functions and compute their SCI scores. The resulting scores show that `DBImpl::Write`, which is not reported by COZ, suffers from interference. As explained in Section 5.5.4, this function inserts new keys in mutual exclusion, which hampers the parallelism.

To conclude, we observe that COZ and ISPOT are complementary. COZ identifies the functions that, when optimized, should lead to better performance. However, these functions do not necessarily suffer from a bottleneck, which makes them difficult to optimize. ISPOT reports a different set of functions: the functions that suffer from interference and degrade performances. Optimizing these functions can substantially improve the application run time.

## 5.8 Comparison with Perfume

In this Section, we compare ISPOT with Perfume [42], another state of the art analysis tool, when running the LevelDB application. Perfume uses inference models over an application logs in order to generate a finite state machine model that describes the application behavior. We reuse the trace generated by ISPOT that is presented in Section 5.7. This trace consists of 1.9 million events, each corresponding to one of the threads entering or leaving one of the most time consuming functions of the application. Perfume gives a graph that depicts the possible succession of events in the trace. The graph also gives the minimum and maximum duration between the events.

First, we evaluate the performance of Perfume and ISPOT. Perfume fails to analyze the provided trace in a reasonable time. We thus provide Perfume with a subset of the trace corresponding to the first 10 000 events generated by one the threads. Perfume analyzes this smaller trace in 5 536 seconds. In comparison, ISPOT analyzes the 1.9 million events traces in 21.2 seconds.

Then, we evaluate the pertinence of the information provided by Perfume and ISPOT. Perfume generates the graph reported in Figure 10. Each node of the graph corresponds to an event in the trace, the edges are the possible event transitions, and the labels on the edges are the measured minimum/maximum transition durations. While this graph shows the global behavior of the application, it does not reveal large variations in the transition durations that could help identify a performance issue. In comparison, ISPOT gives a list of functions sorted by their SCI score. The highest SCI score leads to the interference issue presented in Sections 5.7 and 5.5.4.

We conclude that Perfume and ISPOT are complementary. Perfume helps understanding the general behavior of an application, while ISPOT aims at detecting the functions that suffer from interference.

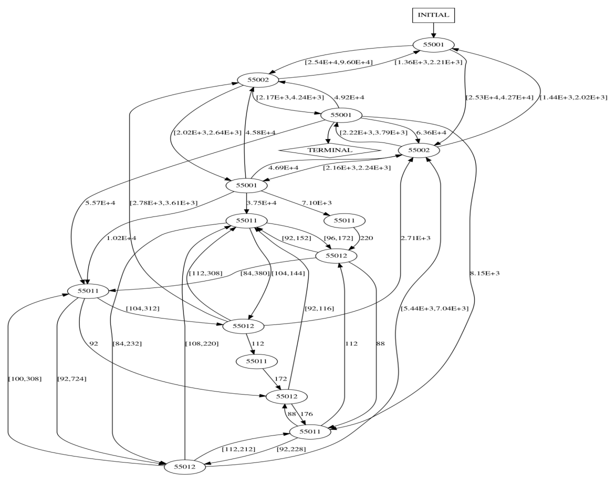


Fig. 10: Graph generated with Perfume

## 5.9 Assessments

Our experiments show that the SCI score accurately assesses how much interference impacts performance, regardless of the causes (lock, parallelism, false sharing, NUMA placement, disk and network). Moreover, our experiment shows that we can remove some of the interference bottlenecks by modifying few lines of code. Finally, we found 2 false positives and we show that they were relatively easy to identify.

## 6 Related work

In this Section, we review existing works that aim at detecting interactions between threads and their impact on application performance. Many projects focus on detecting one type of contention. Collecting hardware counters has been extensively used for detecting cache contention [2], [4], [43], [44], [45], [46], NUMA effects [47], or false sharing [48]. Several works simulate the memory subsystem to detect cache issues [6], or to detect false-sharing [28], [49]. False-sharing can also be detected by tracking the memory access of an application [7], [8], [9], [10], [50]. Hardware sampling has been used for analyzing the cache contention [51], [52], NUMA effects [5], [53], [54], or detecting false sharing [11]. Analyzing the message rate can help detecting network contention [16], [17]. Several works show that lock contention can be detected by measuring the time spent waiting for locks [12], [13], [14], [15], [55]. Overall, these projects define metrics that identify a saturation, but they can not tell if the saturation is the cause of a performance bottleneck. As shown in Section 5, ISPOT comes as a complement to know if removing the saturation should lead to better performance.

Several studies target the optimization of multi-threaded applications in general. In their work, Curtsinger et al. present COZ, a *causal profiler* [40]. COZ identifies the code that should be optimized by slowing down the other parts. As a result, COZ estimates the relative speedup that the developer can expect when the not-slowed down part is optimized. As shown in our evaluation, this work is complementary to ours: COZ identifies the functions that should be optimized while ISPOT identifies the functions that suffer from interference.

Several tools aim at identifying the root cause of a performance defect by using differential execution [18], [21]. Joukob et al. [21] compare the profiles of a function between several runs of the application with different settings (e.g., one process versus many concurrent processes). The function profiles that differ significantly from one run to another are then reported to the developer for further investigation. Song et al. [18] compare good and bad workloads provided by the user in order to identify the root cause, in the code, that leads to different completion times. Stitch [19] is a tool used to automatically identify interactions between large components in a distributed system. This tool is used to identify which component is the root cause of a previously identified performance bottleneck. Recently, Huang et al. [20] propose to use differential execution and performance variation as an indicator to pinpoint the exceptionally *slow* executions of a responsiveness-oriented application. Based on this analysis, they show that they can drastically improve the worst runs. Overall, these works help to find the root cause of a performance defect already identified by the user. ISPOT is complementary and focuses instead on automatically identifying these performance defects for the user when they are caused by interference. Moreover, these works mainly focus on abnormally long executions by comparing the execution times to the average execution time. We show in Section 4.4 that, in case of contention, we should instead focus on the fastest execution time and consider that any longest execution, even average, potentially highlights an interference problem.

As already presented in Section 5.8, Perfume [42] analyzes application logs and uses inference models to build a state machine model that summarizes the application behavior. We show in Section 5.8 that our approach is complementary and has the goal of automatically identifying the interference problems.

A complementary approach to analyze thread interference is proposed by Eyerman et. al [56]. In their work, they use hardware counters to measure which interference impacts performance the most. They measure why and how much interference hampers the scalability of a whole application, but they do not identify where, in the application, the code suffers from interference. In our work, we identify where interference leads to a large slowdown in the code.

## 7 Conclusion

Analyzing the performance of a multi-threaded application is difficult because of the complex interactions between the threads and between the threads and the hardware. In this paper, we propose the SCI score that use differential execution to automatically identify thread interference. Our experiment shows that the SCI score highlights interference, regardless of the interference causes. Thanks to the SCI score, ISPOT successfully detects interference with few and easy to discard false positives. Our experiments also show that, by cross-checking the reports of ISPOT with the reports of classical profiling tools, we can fully understand interference: we can identify the blocks of code that suffers from interference with ISPOT, we can explain why a block of code suffers from interference with complementary tools, and we can measure how much interference degrades the performance of a block of code with ISPOT.

## REFERENCES

[1] "Intel® vtune$^{TM}$ amplifier 2017," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[2] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'03*, 2003, pp. 289.2–.

[3] "Oprofile. a system profiler for linux." [Online]. Available: http://oprofile.sourceforge.net

[4] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proceedings of the 2006 Ottawa Linux Symposium*, 2006, pp. 269–288.

[5] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*, 2012.

[6] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[7] S. M. Günther and J. Weidendorfer, "Assessing cache false sharing effects by dynamic binary instrumentation," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 26–33.

[8] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," in *Proceedings of the international conference on Virtual Execution Environments, VEE'11*, 2011, pp. 27–38.

[9] M. Hobbel, T. Rauber, and C. Scholtes, "Trace-based automatic padding for locality improvement with correlative data visualization interface," in *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'07*, 2007.

[10] T. Liu, C. Tian, Z. Hu, and E. D. Berger, "PREDATOR: Predictive false sharing detection," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'14*, 2014, pp. 3–14.

[11] T. Liu and X. Liu, "Cheetah: detecting false sharing efficiently and effectively," in *Proceedings of the international symposium on Code Generation and Optimization, CGO'16*, 2016, pp. 1–11.

[12] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'10*, 2010, pp. 269–280.

[13] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: A device-driver case," in *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, 2014, pp. 193–206.

[14] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the free-lunch profiler," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, 2014, pp. 291–307.

[15] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'10*, 2010, pp. 739–753.

[16] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[17] M. Casas and G. Bronevetsky, "Active measurement of the impact of network switch utilization on application performance," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'14*, 2014, pp. 165–174.

[18] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, 2014, pp. 561–578.

[19] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'16*, 2016, pp. 603–618.

[20] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'17*, 2017, pp. 64–79.

[21] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating system profiling via latency analysis," in *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'06*, 2006, pp. 89–102.

[22] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalability analysis of spmd codes using expectations," in *Proceedings of the International conference on Supercomputing, ICS'07*, 2007, pp. 13–22.

[23] F. Trahay, Y. Ishikawa, F. Rue, R. Namyst, M. Faverge, and J. Dongarra, "Eztrace: a generic framework for performance analysis," in *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing, CCGRID'11*, 2011, pp. 618–619.

[24] C. Aulagnon, D. Martin-Guillerez, F. Rue, and F. Trahay, "Runtime function instrumentation with EZTrace," in *Proceedings of PROPER 2012 - Workshop on Productivity and Performance*, 2012.

[25] F. Trahay, E. Brunet, M. Mosli Bouksiaa, and L. Jianwei, "Selecting points of interest in traces using patterns of events," in *Proceedings of the International Conference on Parallel, Distributed, and Network-Based Processing, PDP'15*, 2015, pp. 70–77.

[26] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications," in *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*, 2012, pp. 65–76.

[27] M. Scott and W. Bolosky, "False sharing and its effect on shared memory performance," in *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, 1993, p. 57.

[28] T. Liu and E. D. Berger, "SHERIFF: Precise detection and automatic mitigation of false sharing," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'11*, 2011, pp. 3–18.

[29] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Fast and portable locking for multicore architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, pp. 13:1–13:62, 2016.

[30] G. Southern and J. Renau, "Analysis of PARSEC workload scalability," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'16*, 2016, pp. 133–142.

[31] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *Proceedings of the International Symposium on Workload Characterization, IISWC'12*. IEEE, 2012, pp. 59–68.

[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the symposium on High Performance Computer Architecture, HPCA'07*, 2007, pp. 13–24.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the International Symposium on Computer Architecture, ISCA'95*, 1995, pp. 24–36.

[34] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'06*, 2008, pp. 72–81.

[35] D. H. Bailey, "Nas parallel benchmarks," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1254–1259.

[36] M. A. Frumkin and L. V. Shabanov, "Benchmarking memory performance with the data cube operator," NASA, Tech. Rep., 2004.

[37] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[38] M. Zhuang and B. Aker, "memaslap: Load testing and benchmarking a server."

[39] S. Ghemawat and J. Dean, "LevelDB," *URL: http://leveldb.org*, 2011.

[40] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the Symposium on Operating Systems Principles, SOSP'15*, 2015, pp. 184–197.

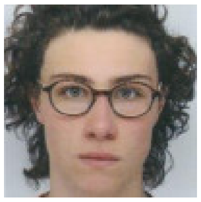
[41] "Facebook rocksdb," https://github.com/facebook/rocksdb/wiki/RocksDB-Basics.

[42] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun, "Mining precise performance-aware behavioral models from existing instrumentation," in *Proceedings of the International Conference on Software Engineering, ICSE'14*, 2014, pp. 484–487.

[43] B. Teabe, A. Tchana, and D. Hagimont, “Application-specific quantum for multi-core platform scheduler,” in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys’16*, 2016, pp. 3:1–3:14.
[44] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, “Contention aware execution: Online contention detection and response,” in *Proceedings of the international symposium on Code Generation and Optimization, CGO’10*, 2010, pp. 257–265.
[45] I.-H. Chung, G. Cong, D. Klepacki, S. Sbaraglia, S. Seelam, and H.-F. Wen, “A framework for automated performance bottleneck detection,” in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS’08*, 2008, pp. 1–7.
[46] C. Xu, X. Chen, R. Dick, and Z. M. Mao, “Cache contention and application performance prediction for multi-core systems,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS’10*, 2010, pp. 76–86.
[47] M. Liu and T. Li, “Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads,” in *Proceedings of the International Symposium on Computer Architecture, ISCA’14*, 2014, pp. 325–336.
[48] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu, “Detection of false sharing using machine learning,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–9.
[49] J. Tao and W. Karl, “Cachein: a toolset for comprehensive cache inspection,” in *Proceedings of the International Conference on Computational Science, ICCS’05*, 2005, pp. 174–181.
[50] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield, “Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing,” in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys’13*, 2013, pp. 141–154.
[51] X. Liu and B. Wu, “Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs,” in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC’15*, 2015, p. 47.
[52] A. Pesterev, N. Zeldovich, and R. T. Morris, “Locating cache performance bottlenecks using data profiling,” in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys’10*, 2010, pp. 335–348.
[53] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, “Traffic management: A holistic approach to memory placement on numa systems,” in *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS’13*, 2013, pp. 381–394.
[54] F. Trahay, M. Selva, L. Morel, and K. Marquet, “NumaMMA: Numa MeMory Analyzer,” in *Proceedings of the International Conference on Parallel Processing, ICPP’18*, 2018.
[55] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, “Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior,” in *Proceedings of the International Symposium on Computer Architecture, ISCA’13*, 2013, pp. 511–522.
[56] S. Eyerman, K. Du Bois, and L. Eeckhout, “Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS’12*, 2012, pp. 145–155.

**Mohamed Said Mosli Bouksiaa** is a PhD student at Télécom SudParis since 2014. His research focuses on automatic performance analysis for high performance computing applications.

**François Trahay** is an associate professor at Télécom SudParis since 2011. He is a member of the distributed systems research group of the computer science department. He received his Ph.D. degree in computer science from the University of Bordeaux in 2009. He has been working on runtime systems and performance analysis for high performance computing since 2006.



**Alexis Lescouet** is a PhD student at Télécom SudParis since 2017. His research focus on system virtualization for NUMA machines.

**Gauthier Voron** is a PhD student at UPMC Sorbonne Université since 2014. His research aims at understanding and improving the performance and the design of distributed or concurrent systems. Especially, his interests include blockchains, NUMA architectures and system virtualization.

**Rémi Dulong** is a PhD student from Télécom SudParis and the University of Neuchâtel since 2018. His research focuses on the performance of NVMRAM for large applications running on NUMA systems.

**Amina Guermouche** is an associate professor at Télécom SudParis since 2016. She received her Ph.D. degree in computer science from the Paris-Sud university under the supervision of Franck Cappello at LRI. Her research interest focus on energy minimization and fault tolerance on large scale platforms.

**Élisabeth Brunet** is an associate professor at Télécom SudParis in the distributed systems research group since 2010. She defended her PhD thesis in 2018 within the INRIA Runtime team of LaBRI under the direction of Raymond Namyst. Her research interests are in high performance computing in the cluster architecture landscape.

**Gaël Thomas** is a full professor in the computer science department of Télécom SudParis where he leads the distributed systems research group. He his interested by virtualization, operating systems, concurrency and language runtimes, with a focus on performance and safety. Before joining Telecom SudParis in 2014, he was associate professor from 2006 to 2010 at UPMC Sorbonne Université in the LIP6 laboratory.