Identifying thread interference with performance variation

Abstract

Understanding the performance of a multi-threaded application is difficult. The threads interfere when they access the same hardware resource, which slows down their execution. Unfortunately, current profiling tools are unable to identify the most problematic interference, because they cannot classify interference on different hardware resources. In this paper, we propose a holistic metric able to simultaneously classify interference on different hardware resources. The metric considers performance variation as a universal indicator. We propose an interactive profiling tool to compute this metric. With an evaluation of 27 applications we show that our profiling tool successfully identify 10 performance bottlenecks caused by 6 different kinds of interference.

1. Introduction

Understanding the performance of a multi-threaded application is a nightmare. Threads regularly interfere because they simultaneously access the same shared hardware resource (e.g., a cache, a memory controller or a network card), which decreases the parallelism and thus the performance of the application. As an interference remains often implicit in the code, the developer uses profiling tools to detect it. Typically, current profiling tools can identify contended locks [43, 49, 13, 4], falsely-shared cache lines [22, 19, 50, 20, 29, 28], saturated network stacks [5, 9] or overloaded memory controllers [1, 14, 3, 15]. Unfortunately, these tools give incomparable metrics and the developer cannot easily know which interference hampers the application the most. Hence, the developer often tries to remove a randomly chosen interference, identified by one of the profiling tools, without even knowing if removing the interference can actually improve performance.

As an alternative, several tools propose to analyze the temporal behavior of the application. These tools identify unusual executions by using explicit thresholds [49], by comparing runs with different settings (number of threads, workload) [24], or with mining techniques [33, 35]. None of these techniques is satisfying. Explicit thresholds are hard to identify and they are often machine-specific. Running an application with different settings often changes the executed code, making the comparison between the runs difficult to interpret. Mining techniques identify unexpected behavior by considering that the average behavior is the normal behavior, but they are unable to tell if this average behavior is efficient. As a result, today, although we can identify the slow component of a large distributed system [51, 33], we still do not have a satisfying tool to identify the most problematic interference of a multi-threaded component.

In this paper, we propose a new holistic metric able to identify any interference problem. This metric identifies interference problems regardless of the contended hardware resource. For that purpose, our metric considers that performance variation is the indicator of any interference problem. It relies on temporal behavior, but, instead of focusing on the average behavior, it focuses on the exceptionally fast executions, even if they occur very rarely. Our idea starts with a simple intuition. An exceptionally fast execution of a sequence of instructions is probably an execution that was not slowed down by interference from the other threads. Thus, we take this exceptional execution as a reference. Next, we consider that a slower execution is caused by interference, and we compute the percentage of time lost by a thread because of interference during the execution of the sequence. We define this relative slowdown as the RDAM score (Relative DistAnce to Minimum) of the sequence. It directly indicates the potential improvement of the thread if the sequence of instructions does not suffer interference anymore.

As the RDAM score gives the potential improvement, it naturally indicates which sequence of instructions suffers interference the most, regardless of the contended hardware resource. Moreover, computing the RDAM score does not require a-priori knowledge because we use the fastest execution as a reference. The RDAM score also requires a single run, and compares the times taken by a sequence of instructions with a single setting. Finally, unlike standard deviation or mining techniques, which ignore the fastest execution time in favor of the average execution time, the RDAM score is able to identify an interference problem when the average behavior is inefficient.

The RDAM metric comes as a complement to the existing profiling tools. The RDAM score gives the overhead caused by interference on a sequence of instructions, but does not explain alone why the sequence suffers interference. In a complete performance debugging process, a developer can thus start by computing the RDAM scores and then uses existing profiling tools to explain the most problematic interference reported by the RDAM metric. The developer can also start with existing profiling tools, and then compute the RDAM score of a bottleneck reported by a tool in order to verify that the bottleneck actually hampers performance.

In addition to the RDAM metric, we propose an interactive toolchain that helps the developer to identify the high RDAM scores. We evaluate the usefulness of the RDAM metric with 27 applications (7 from the Splash2 benchmark [47], 7 from the Phoenix-2 benchmark [37], 4 from the Parsec benchmark [6], 7 from NAS Parallel Benchmarks [2], memcached [16]...
and LevelDB [18]). Our evaluation shows that the RDAM metric can identify and classify different interference causes, with few and easily identifiable false positives. Thanks to the RDAM metric, we identify both well-known and new interference issues. They are caused by false sharing, lock contention, poor parallelism, memory placement, network stack and disk I/O in real applications. In detail, we found that:

- The RDAM metric is able to detect interference in 13 functions from 11 applications. Among the 13 functions, 3 are false positive. The others, from 9 applications, pinpoint actual interference problems. 6 interference problems were previously identified in other works, while 4 are new.
- Thanks to the RDAM metric, we are able to remove 7 performance bottlenecks from 6 applications, by modifying at most only 25 lines of code in each application.
- We show that the 3 false positives appear when the performance variation is caused by a varying workload but not by an interference. We show that, even if a manual analysis of the source code is required, these false positives are easy to identify and to discard.

The paper is organized as follows. Section 2 formally presents the RDAM metric, while Section 3 discusses how we can practically compute the RDAM metric and Section 4 presents the interactive toolchain used to compute the RDAM metric. We present an evaluation of the RDAM metric with micro-benchmarks in Section 5, and with applications in Section 6. Finally, Section 7 presents the related work and Section 8 concludes the paper.

2. The RDAM metric

The RDAM metric pinpoints the repetitive sequences of instructions that suffer interference at runtime. Intuitively, the RDAM metric gives the speedup that the developer can expect if a sequence of instructions does not suffer interference anymore. Figure 1 illustrates this intuition. Each box represents a sequence of instructions executed by a thread, and the gray boxes are repetitive sequences of instructions. The RDAM of the gray boxes at the top of the figure is the performance improvement (represented at the bottom) that would be obtained if all the occurrences of this sequence of instructions had the minimum duration (in this case: 22).

In this section, we provide the formal definition of the RDAM metric. Then, we justify the definition of the RDAM metric by showing that the slowdown of a thread caused by interference is the sum of the RDAM scores of the different repetitive sequences of instructions. Finally, we discuss the experimental conditions required to be able to compute the RDAM score.

2.1. Formal definition

As presented in Figure 1, computing the RDAM score starts by computing the execution time of the repetitive sequences of instructions (the gray and white boxes). For that purpose, we model the run of a thread as the history of the instructions executed by the thread. Then, we identify repetitive sequences of instructions in this history. We present a heuristic able to identify sequences with many repetitions in Section 2.4.1.1. In this section, we are just interested in formalizing them. Formally, we can rewrite the history as a sequence of sequences of instructions. We note this history \( \mathcal{H} = \{s_i\} \), in which \( s_i \) is the \( i \)-th sequence of instructions executed by the thread.

In this history, we say that two sequences \( s_i \) and \( s_j \) are equivalent, and we note \( s_i \sim s_j \), if they contain the same instructions executed in the same order. In Figure 1, the gray boxes represent equivalent sequences of instructions. We finally define a set of repetitive sequences of instructions \( R_i \) as the set of sequences that are equivalent to \( s_i \): \( R_i = \{ s_j | s_j \sim s_i \} \).

In Figure 1, the set of gray boxes form an \( R_i \); \( R_i = \{ s_1, s_4, s_5, s_9, s_{10} \} \).

We assess the cost of interference on a repetitive sequence of instructions by considering that two equivalent sequences of instructions \( s_i \) and \( s_j \) should have the same execution time in absence of interference. More formally, if we note \( \overline{d}_i \) the execution time of \( s_i \), we consider that \( s_i \sim s_j \implies \overline{d}_i = \overline{d}_j \) in absence of interference. Technically, if \( s_i \sim s_j \) and if \( s_j \) takes more time than \( s_i \), we suppose that another thread slows down \( s_j \) because it accesses the same hardware resources. If we note \( \overline{d}_i \) the ideal execution time of \( s_i \), i.e., the execution time of \( s_i \) in absence of interference, then we consider that \( \overline{d}_i - \overline{d}_j \) is the overhead caused by interference during the execution of \( s_i \).

Finally, if we take a set of repetitive sequences \( R_i = \{ s_i \} \), the overall overhead caused by interference during the execution of the \( \{ s_i \} \) is equal to \( \sum_{j \in R_i} (\overline{d}_j - \overline{d}_i) \). As \( s_i \sim s_j \implies \overline{d}_j = \overline{d}_i \), this overhead is then equal to \( (\sum_{j \in R_i} \overline{d}_j) - n_i \overline{d}_i \), in which \( n_i \) is the number of occurrences in \( R_i \). The RDAM score of \( R_i \) simply makes this overhead relative to the execution time of the thread:

\[
RDAM_i = \frac{(\sum_{j \in R_i} \overline{d}_j) - n_i \overline{d}_i}{T}.
\]

2.2. Justification of the RDAM metric

The RDAM metric perfectly captures the slowdown caused by interference. Formally, if we consider an execution time
of $T$, and if we imagine an ideal execution time of $T$ without interference, the overall slowdown caused by interference is equal to $(T - T)/T$. This slowdown is equal to the sum of the RDAM scores of the repetitive sequences of instructions:

$$
\frac{T - T}{T} = \frac{(\sum_{i=1}^{N} \sum_{j \in R_i} d_j) - (\sum_{i=1}^{N} \sum_{j \in R_i} \overline{d}_j)}{T} = \sum_{i=1}^{N} \left( \frac{N}{T} \right) \left( \sum_{j \in R_i} d_j - \overline{d}_j \right) = \sum_{i=1}^{N} \left( \frac{N}{T} \right) d_j - n_i * \overline{d}_j = \sum_{i=1}^{N} RDAM_i
$$

This result shows that the RDAM score is exactly the metric that captures the slowdown of the thread caused by interference on a given repetitive sequence of instructions. For a high score, if the developer can find a way to remove interference, then the score directly indicates the potential improvement of the thread.

### 2.3. Computing the RDAM

In order to compute the RDAM of $R_i$, we have to compute (i) $d_j$, i.e., the execution times of each $s_j \in R_i$, (ii) $n_i$, i.e., the number of occurrences in $R_i$, and (iii) $\overline{d}_i$, i.e., the ideal execution time of $s_i$ without any interference. Computing $d_j$ is straightforward by instrumenting the application. We can also easily find the number of sequences $n_i$. However, computing $\overline{d}_i$ is much more difficult. It would require an execution of a single thread in isolation, which is not always feasible because the code executed by each thread may vary a lot when we change the number of threads.

In order to compute $\overline{d}_i$, we rely on statistics. We suppose that if the number of occurrences is large, then often, at least one occurrence executes without interference or with few interference. This hypothesis, if verified, solves our problem, because we can consider that the fastest execution is an interference-free execution, i.e., that $\overline{d}_i = \min_{j \in R_i} \{d_j\}$.

Practically, as we don’t know $\overline{d}_i$, we cannot prove that the fastest execution is an interference-free execution. However, we often measure tens of thousands to millions of occurrences and we experimentally show in Section 5 that the probability of executing an (almost) interference-free occurrence is large in this case.

### 3. From theory to practice

From theory to practice, there is a gap. In this Section, we present how we can practically implement an interactive toolchain able to compute the RDAM score and we highlight the limitations of our approach.

In our theoretical model, we suppose that we can record a timestamp before the execution of each instruction, and that we can identify the repetitive sequences in this trace. However, we have measured that recording a timestamp costs around 100 cycles (50ns). Instrumenting each instruction would slow down the application drastically, which would make the RDAM score inaccurate. In order to decrease this overhead, we have to directly measure the execution time of larger sequences of instructions. However, this strategy introduces false positives, because the execution time of a large sequence of instructions may vary with the loops and conditional statements.

As finding a perfect trade-off that minimizes both the number of false positives and the overhead is difficult in general, we propose an interactive toolchain driven by the developer, which has the knowledge of the application. By default, the toolchain considers that a function is a repetitive sequence of instructions, regardless of the arguments. We have chosen the function by default because automatically instrumenting a function is relatively straightforward and gives interesting results for 70% of our evaluated applications (see Section 6). We have chosen to ignore the parameters by default, because considering the parameters drastically decreases the number of samples per repetitive sequence of instructions, which in turn drastically decreases the probability of capturing an interference-free execution.

Profiling functions is, however, far from perfect and can lead to false positive. For this reason, the developer can tune the toolchain in two different ways described in the remainder of this section.

### 3.1. Manual instrumentation

Measuring the RDAM at the level of a function is not always the best option because a function is sometime too coarse or too fine. A function with nested loops executed few times, for example, is too coarse, because the probability of capturing an interference-free execution is low. Manually instrumenting an inner loop gives a more accurate RDAM.

In an event-oriented application, such as memcached, instrumenting the functions is too fine, because interference can come from a large time spent in the event queue, which is not captured if we instrument functions. In this case, measuring an end-to-end processing time that involves several event handlers is more interesting.

As we cannot predict which probes gives the most interesting RDAM in general, we have chosen to let the developer manually instrument an application when instrumenting a function is inadequate. In this case, the developer can put a probe at any place in the code by adding a single line of code.

### 3.2. False positives and parameter-dependent functions

When an instrumented sequence of instructions is not repetitive, our toolchain reports a false positive. In this case, the time variation is not caused by interference, but by a varying workload. The workload can change because the function is parameter-dependent, i.e., because its workload varies a lot.
when we change its parameters (e.g., `memcpy`). The workload can also change for other reasons, e.g., because the sequence of instructions access a global variable such as a linked list, or simply because of the warm up of the caches. As in our evaluation, the 3 false positives are caused by parameter-dependent functions, we only focus on this case.

Perfectly handling the parameter-dependent functions is difficult because predicting which parameter matters is often application-specific. For example, if a function only writes small chunks in a file, the size of the chunks does not change the execution time of the function, while this is not the case if the size changes a lot. We faced this case in the DC application, for which it is more interesting to ignore the size of the chunks in order to highlight a bottleneck in the kernel stack.

As handling parameter-dependent functions in general is difficult, we let the developer interactively specifies which parameters change the behavior of a function. Moreover, our toolchain uses a default list of well-known parameter-dependent functions, such as `pthread_mutex_lock`. With this strategy, in the 27 evaluated applications, our toolchain reports 3 false positives discussed in detail in Section 6.4. For these false positives, the developer has to interactively select the parameters that matter.

### 4. Interactive profiling toolchain

This section presents the interactive toolchain used to compute the RDAM scores. As presented in Figure 2, we split the analysis in two phases. During the first in-vitro profiling phase, the toolchain runs an instrumented version of the application in order to gather the execution times of the repetitive sequences of instructions. The toolchain can either automatically instrument the time-consuming functions of the application, or let the developer manually instrument the application. As presented in Section 3.1, manual instrumentation is required when profiling functions is either too fine or to coarse. As stated in the introduction, manual instrumentation can also be useful to instrument a sequence of instruction pinpointed by another profiling tool, in order to verify that the reported bottleneck actually hampers performance.

During the second phase, the toolchain first identifies the repetitive sequences of instructions in the trace generated during the first phase, and then computes the RDAM score of each repetitive sequence of each thread.

#### 4.1. In-vitro profiling phase

The toolchain generates the traces during an in-vitro profiling phase. This phase aims at collecting the timestamps used to compute RDAM score. When the developer lets the toolchain automatically instrument the application, the toolchain only records the timestamps for the functions that take a significant time because recording a timestamp for each function can lead to a large slowdown that can change the application behavior.

In order to identify these time-consuming functions, the toolchain runs the application a first time with the `linux perf` tool (1. Sampling in Figure 2). `Linux perf` uses a sampling technique to identify the time spent in any function. Then, the toolchain selects the functions that take a significant part of the total execution (specified by the developer) and ignore the other functions as, in most of the cases, improving their performance should not significantly improve the overall performance.

After that, the toolchain runs the application with the `EZTrace` tracing framework [46], which automatically instruments the interesting functions reported by `linux perf` (2. Instrumentation in Figure 2) and records the timestamps in a file (3. Trace generation in Figure 2). We detail these two steps in the remainder of the section.

##### 4.1.1. Instrumenting an application

Since manually instrumenting an application can be a tedious task for the developer, we use an automatic tool to instrument a binary application. The tool takes the output of `linux perf` and finds the symbols of the time consuming functions in the binary of the application (using the Dwarf format). Then, it extracts the function prototypes from the debugging symbols, and gives these prototypes to `EZTrace`.

`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, while if the function is located in the binary, `EZTrace` intercepts the calls by patching the binary when the binary is loaded in memory by the ELF loader. In both cases, `EZTrace` records an event (marking the beginning of the function), calls the original function, and records an event (marking the end of the function).

##### 4.1.2. Generating the trace

As presented in Figure 2 (3. Trace generation), `EZTrace` records the events using LiTL (Lightweight Trace Library) [21]. Each event consists of a timestamp (the CPU cycle counter), an event id (the function name), and additional optional parameters used to store the function parameters. Each thread of the application stores its events in its own preallocated buffer so as to avoid thread synchronizations. When a buffer is full, LiTL flushes the buffer to disk and reinitializes the buffer. LiTL also flushes the buffers to disk at the end of the application. We ensure that the buffers are rarely flushed during the run by using large buffers.
On our small machine (Xeon4, see Section 5), we pre-allocate 1 GB for the buffers, while on our large machine (Opteron48, see Section 5), we pre-allocate 32 GB.

During the run, the instrumentation overhead is reduced to the cost of an additional function call (from the EZTrace wrapper to the instrumented function), two accesses to the timestamp counter (tsc), the copy of few bytes in the buffer, and, only when EZTrace modifies a binary, two additional jmp instructions. As a result, on the machines used for the evaluation (see Section 5), we measured that the overhead always remains below 100 ns.

4.1.3. Recording the call stacks. The generated traces contain events associated to the entry and exit points of the functions. In order to identify why a function has a high RDAM score, the developer is often interested by the call stack of the function. This is, for example, the case with the pthread_mutex_lock function: in this case, the developer wants to know which critical section is protected by the lock (underlined) is a repetitive sequence that appears twice. We consider four different micro-benchmarks. The first and second micro-benchmarks exhibit a problem that occurs in our case. It tries to find this sequence 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.

As soon as RDAMcalculator identifies the repetitive sequences of instructions, it computes the RDAM score of each repetitive sequence by considering that the occurrence with the minimum duration is interference-free (see Section 2.3). The tool then reports the repetitive sequences along with their RDAM score and the recorded stack traces. The developer can then interactively changes how RDAMcalculator handles the parameter-dependent functions and can discard the false-positives before restarting the computation of the RDAM scores.

5. Micro-benchmark evaluations

In this section, we study the RDAM scores of several simple micro-benchmarks that implement known interference problems. This first study has two different goals.

First, this evaluation has the goal of showing that the RDAM score is actually correlated to an interference problem. For that purpose, we evaluate well-known problems: lock contention (blocking and non-blocking locks), false sharing, and I/O contention. These problems are frequent performance problems that are caused by thread interference. In each micro-benchmark, we vary the frequency of the interference. We then compute the RDAM score of the sequences of instructions affected by interference and we verify that it is correlated to the performance of the micro-benchmark.

Second, as presented in Section 2.3, we suppose that by recording a sequence of instructions with many repetitions, the probability of recording an interference-free occurrence is high. Hence, this experiment has also the goal of verifying that this hypothesis is correct.

For our evaluations, we use two machines: (i) Xeon4 has 4 cores, 8 GB of memory, 1 Intel Xeon E5-2603 socket, 1 NUMA node. Linux version: 4.4.0-1-amd64, gcc version 5.3.1, glibc version: 2.22-4, and (ii) Opteron48 has 48 cores, 256 GB 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.

5.1. Summary of the micro-benchmarks

We consider four different micro-benchmarks. The first and the second micro-benchmarks exhibit a problem that occurs when multiple threads try to acquire the same lock at the same time. In case of contention, the lock acquisition algorithm itself collapses [30], which leads to drastically degraded performance.
The first thread \((\text{my\_rank} = 0)\) continuously updates its variable \(x\). The other thread \((\text{my\_rank} = 1)\) 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.

The fourth micro-benchmark exhibits an interference caused by contention on the I/O stack to access a disk. When many threads perform I/O operations simultaneously on a disk, one of the components of the I/O stack (e.g., the disk itself, the I/O controller or the I/O subsystem of the operating system) can saturate. In this case, we have a typical case of overloaded resource, which leads to thread interference and performance degradation.

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 \texttt{O\_DIRECT} flag, which ensures that every call to \texttt{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.

5.2. Analysis of the micro-benchmarks

Figures 3, 4, 5 and 6 report the evaluation of the micro-benchmarks. In each figure, \(a\) gives the performance of the micro-benchmark when we vary delay (solid line) along with the occurrence with the minimal duration (dotted line). The \(b\) figure gives the RDAM score when we vary delay. For the two lock micro-benchmarks, \(a\) 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 solid lines in 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 RDAM score seems to behave exactly as the completion time: when the delay increases, the RDAM scores also decreases.

In order to confirm this observation, we compute the Pearson product-moment correlation coefficient between the completion time and the RDAM 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. When this number is close to -1 or 1, it means that a linear relation exists between the two variables. Table 1 reports the correlation coefficient between the completion time and the RDAM score for each micro-benchmark, along with the number of samples (points on the x-axis) for each variable. We can observe that the co-
efficient is high in all the experiments (above 0.95), which confirms that a linear relation between the completion time and the RDAM score exists in the micro-benchmarks. From this strong correlation, we can conclude that the RDAM score captures the performance degradation caused by interference in the micro-benchmarks.

Moreover, when we observe the dotted lines in the (a) figures, we can see that the occurrence with the minimum completion time remains constant for each contention. This result shows that we are actually able to capture at least one interference-free occurrence of the repetitive sequence in each of the micro-benchmarks, even when the contention level is extremely high. This validates the hypothesis formulated in Section 2.3: when the number of occurrences of a repetitive sequence is large, the probability of having an interference-free occurrence seems to be large.

### 6. Applications evaluation

This second evaluation has the goal of verifying that the RDAM metric can capture performance bottlenecks caused by interference in real applications. After a presentation of the applications, this section reports the time-consuming functions of the applications, the RDAM scores of the time-consuming functions and a systematic analysis of these scores. Finally, this section presents a detailed analysis of the false-positives reported by our toolchain.

#### 6.1. Evaluated applications

We evaluate 27 applications summarized in Table 2. We have selected these applications because they are widely used to evaluate the parallelism.

Phoenix-2 [37], a MapReduce for shared-memory systems written in C. It comes with small sample applications with data sets ranging from 59 to 512 MB. The Splash2 benchmark [47] contains small multi-threaded C applications, ranging from a ray tracer to a large-scale ocean movement computations. The Parsec 2.1 benchmark [6] contains small and large multi-threaded C++ applications from various fields such as financial analysis or data-mining. NPB (NAS Parallel Benchmark 3.3 [2]) contains moderate to large Fortran and C applications, ranging from linear algebra to a data mining application, which writes 2.5 GB of data in the file system [17]. We use the large class C dataset. While the other benchmarks synchronize with POSIX locks and condition variables, NBP synchronizes through OpenMP. Memcached 1.4.36 [16] is an in-memory cache widely used for web servers. We evaluate memcached with the memaslap client [52], 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 memslap on another Xeon4 machine. LevelDB 1.20 [18] 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.

#### 6.2. Hotspots identification

As presented in Section 4.1, our profiling toolchain first uses `linux perf` in order to identify the time-consuming functions. Table 3 presents the most time-consuming functions of the benchmarks. The last column of Table 3 also reports the overhead caused by instrumentation. In the worst case, the overhead remains below 30 %. This overhead remains reasonable for an in-vitro profiling and should not drastically change the behavior of the applications.

Our profiling toolchain then automatically instruments the
functions reported in Table 3, except for the Phoenix-2 benchmark and the memcached application. In Phoenix-2, the pinpointed functions are systematically called only once, which is not enough to compute an accurate RDAM score (see Section 2.3). These functions execute large loops and we have thus manually inserted the calls to the EZTrace wrapper function in the loop bodies. Memcached is an event-oriented application and relies on libevent. As presented in Section 3.2, measuring the RDAM of the function is too fine to exhibit an interference problem. We have thus manually instrumented multiple events in memslap (the client side) and in memcached (the server side) in order to compute the completion time of a request from both sides.

6.3. Analysis of the RDAM scores

Figure 7 reports the highest RDAM score identified by our toolchain for each application identified in the first step. We identify 13 functions with a high RDAM score. We use an arbitrary threshold of 0.2, because we think that optimizing an application that loses less than 20% of its time because of interference becomes probably useless. We exhaustively present the 13 functions in the remainder of the section.

6.3.1. Lock contention We found two functions with a high RDAM score caused by lock contention, i.e., for which the lock algorithm itself collapses. One comes from Raytrace and has an RDAM score of 0.82, the other from Radiosity and has an RDAM score of 0.87. Listing 4 illustrates the problem with the critical section associated with the lock acquisition of Raytrace. These critical sections were also identified by Lozi et al. [30, 31], and we have thus reused their algorithm (the RCL lock). We confirm their result: using an RCL lock divides by 4.81 the completion time of Raytrace and by 5.37 the completion time of Radiosity. We identify two functions that suffer false-sharing, one from Raytrace that was never reported previously by Liu et al. [27] (RDAM of 0.37).

The high RDAM in Raytrace appears after correcting the lock contention (see Section 6.3.1), 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, while the RCL lock should ensure that the cache line containing gm->rid should remain in the cache of the RCL server [30]. As an unexpectedly high number of cache misses is often caused by false sharing, 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 6.3.1), 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 (RDAM of 0.25), the other from Linear_regression that was previously reported by Liu et al. [27] (RDAM of 0.37).

Table 2: Summary of the evaluated applications

<table>
<thead>
<tr>
<th>Hardware setting</th>
<th>Phenix-2</th>
<th>Splash2</th>
<th>Parsec</th>
<th>NPB (C)</th>
<th>Memcached</th>
<th>LevelDB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># threads</td>
<td>Xeon4</td>
<td>Opteron48</td>
<td>Opteron48</td>
<td>Opteron48</td>
<td>2*Xeon4</td>
<td>Xeon4</td>
<td></td>
</tr>
<tr>
<td># applications</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td># manually instrumented</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td># applications with a high score</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td># functions with a high score</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2^172</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td># false positives</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td># applications with true defects</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Discussed in Section</td>
<td>6.3.2</td>
<td>6.3.1, 6.3.3</td>
<td>6.3.3</td>
<td>6.3.5, 6.3.6</td>
<td>6.3.4</td>
<td>6.3.3</td>
<td></td>
</tr>
<tr>
<td># issues never reported</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td># corrected functions</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Listing 4: code of the hottest function in Raytrace

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

Figure 7: Highest RDAM scores
Thanks to these optimizations, we have eliminated the high RDAM scores in both applications.

### 6.3.3. Parallelism

We have identified a (new) problem of parallelism in LevelDB. The function `pthread_cond_wait` has a high RDAM score of 0.66. After an analysis of the code, we identified a scalability bottleneck, because the write operations execute in mutual exclusion when the application inserts new keys. When a thread of LevelDB inserts a new key, it enters a monitor (with a variable condition) for writing the transaction. When another thread inserts a key, it uses the monitor in order to wait for the completion of the previous write. As performing a write is slow, the writes in mutual exclusion hamper the parallelism. The RDAM 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. We can, however, confirm our observation by measuring the scalability of LevelDB. We have measured that the duration of operations quickly increases when the number of threads increases (2.4 micros/op with 1 thread, 8.8 micros/op with 2 threads, 20.1 micros/op with 4 threads and 47.8 micros/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` has a high RDAM score of 0.99 (already reported in [42, 38]). The execution trace shows that the 48 threads of the application synchronize repeatedly with this barrier function. The threads of the application thus spend most of their time waiting for the other threads. The RDAM score is high because in some rare cases a thread traverses the barrier quickly. Correcting this bottleneck would require a large code rewriting.

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

### 6.3.4. Network contention

We identify a network bottleneck between the memcached server and the memslap client with our setting. The server handles 36,265 transactions per second (TPS) with a network rate of 11.4 M/s. We observe a high RDAM score of 0.93 to process a request at the client side, while the score at the server side remains below 0.1. This result shows that the network saturates. After some investigations, we have (unexpectedly) discovered that the client machine was only able to configure the 1G-network card at 100M because of a flaw in the network cable. This typical network issue was easily pinpointed by the RDAM metric.

After having simply replaced the cable, the server handles 193,716 TPS, for a network rate of 61.1 M/s and a RDAM score of 0.82 at the client side (the RDAM at the server side 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.

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Function</th>
<th>% of time</th>
<th>Instrum. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsec-2</td>
<td>histogram</td>
<td>calc_hist</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>kmeans</td>
<td>find_clusters</td>
<td>79%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>calc_means</td>
<td>21%</td>
</tr>
<tr>
<td></td>
<td>linear_regression</td>
<td>linear_regression_pthread</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>matrix_multiply</td>
<td>matrixmult_map</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>pca</td>
<td>calc_cov</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>string_match</td>
<td>string_match_map</td>
<td>88%</td>
</tr>
<tr>
<td></td>
<td>word_count</td>
<td>__strcmp_sse2_unaligned</td>
<td>44%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__workcount_reduce</td>
<td>27%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__memmove_sse3_back</td>
<td>22%</td>
</tr>
<tr>
<td>Barnes</td>
<td>hackcomf</td>
<td>65%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>walk</td>
<td>12%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subdvp</td>
<td>12%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23.18%</td>
</tr>
<tr>
<td>FMM</td>
<td>__ListInteraction</td>
<td>__ListInteraction</td>
<td>37%</td>
</tr>
<tr>
<td></td>
<td>DownwardPass</td>
<td>DownwardPass</td>
<td>32%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24%</td>
</tr>
<tr>
<td>Ocean cont.</td>
<td>pthread_barrier_wait</td>
<td>pthread_barrier_wait</td>
<td>23%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>relax</td>
<td>14%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>slave2</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__lll_lock_wait</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__lll_unlock_wait</td>
<td>11%</td>
</tr>
<tr>
<td>Ocean non cont.</td>
<td>__lll_unlock_wait</td>
<td>__lll_unlock_wait</td>
<td>21%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lapacalc</td>
<td>16%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>relax</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pthread_barrier_wait</td>
<td>11%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.39%</td>
</tr>
<tr>
<td>Radiosity</td>
<td>__lll_lock_wait</td>
<td>__lll_lock_wait</td>
<td>37%</td>
</tr>
<tr>
<td></td>
<td>__process_task_wait_loop</td>
<td>__process_task_wait_loop</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21%</td>
</tr>
<tr>
<td>Raytrace car</td>
<td>__lll_unlock_wait</td>
<td>__lll_unlock_wait</td>
<td>25%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__lll_lock_wait</td>
<td>31%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swaptions</td>
<td>__HM_SimPath Forwards</td>
<td>__HM_SimPath_Forward</td>
<td>33%</td>
</tr>
<tr>
<td></td>
<td>fluidanimate</td>
<td>ComputeForcesMT</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td>ComputeDensitiesMT</td>
<td>ComputeDensitiesMT</td>
<td>32%</td>
</tr>
<tr>
<td></td>
<td>facesim</td>
<td>Add_Forces</td>
<td>29%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>__Add_Force_2D</td>
<td>__Add_Force_2D</td>
<td>17%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update_Position</td>
<td>44%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>compute_rhs</td>
<td>compute_rhs</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x_solve</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y_solve</td>
<td>13%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z_solve</td>
<td>13%</td>
</tr>
<tr>
<td></td>
<td>CG</td>
<td>compute_grad</td>
<td>4%</td>
</tr>
<tr>
<td>DC</td>
<td>KeyComp</td>
<td>MultiWayMerge</td>
<td>26%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__memcpp_sse2</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write_noacce</td>
<td>19%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13%</td>
</tr>
<tr>
<td>EP</td>
<td>__sec754Log_sse2</td>
<td>__sec754Log_sse2</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td>__sec754Log_sse2</td>
<td>__sec754Log_sse2</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vranlc</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__write</td>
<td>35%</td>
</tr>
</tbody>
</table>

**Table 3: Time consuming functions reported by Linux `perf`**

**In streamcluster, the `parsec_barrier_wait` has a high RDAM score of 0.99 (already reported in [42, 38]). The execution trace shows that the 48 threads of the application synchronize repeatedly with this barrier function. The threads of the application thus spend most of their time waiting for the other threads. The RDAM score is high because in some rare cases a thread traverses the barrier quickly. Correcting this bottleneck would require a large code rewriting.**
remains below 0.1). This high RDAM score shows that the network remains a bottleneck. We confirm this hypothesis by co-localizing the server and the clients on the same machine (a 32-core Intel Xeon E5-1607). We measure a lower RDAM score of 0.62 for a network rate of 632.4 M/s (2,036,946 TPS).

This high RDAM score at the client side, coupled with a RDAM score below 0.1 at the server side, shows that even when we co-localize the server and the client on the same machine, the local network loop remains the bottleneck.

We have also tested 25 other settings to try to exhibit a high RDAM score at the server side. In all our experiments, the RDAM score at the server side remains below 0.1, which suggests that memcached does not suffer any interference, probably because the network always saturates first.

6.3.5. Memory placement We have identified a (new) problem of NUMA memory placement in the LU application. The profiling phase shows that LU spends 50% of its time in sync_left and sync_right, and 22% in rhs. Five other functions take approximately 5% of the execution time each.

By instrumenting the eight functions, our toolchain generates a trace with 30.8 million events. The analysis of the 1.3 GB trace by our toolchain highlights three functions with a high score: sync_left (RDAM of 0.75), blts (0.41), and buts (0.34). The other functions have a score lower than 0.03.

A quick analysis of the source code of sync_left shows that this function synchronizes the OpenMP threads, such that thread \(i\) waits until thread \(i-1\) terminates its task. This leads to a cascade effect where each thread waits for the previous thread before starting its computation. Note that the source code also shows that blts and buts both perform linear algebra operations on a matrix and call sync_left. Their high score is caused by the score of sync_left.

The generated trace also shows that some of the threads run the blts and buts functions much faster than the others. In order to explain this behavior, we measured the cache misses by generating a new trace that also includes PMU (Performance Monitor Units). Analyzing this trace shows that the threads that run faster generate less cache misses than the others (approximately 40 cache misses per execution of blts versus 2,000 cache misses for slower threads). This leads us to the conclusion that the problem is caused by a NUMA effect: the matrix is located on a NUMA node, which causes the threads that execute on the other NUMA nodes to run slowly. Because of the cascade effect, the slower threads delay the other threads and degrade the performance. To solve this performance issue, we bind the threads using hwloc [7], in order to make the first threads run faster. The results that we obtain show that the thread binding reduces the cascade effect and improves the performance of the application by 28%. However, the cascade effect still exists, which hampers parallelism, and the RDAM scores of sync_left, blts, and buts remain high (respectively 0.58, 0.32, and 0.27).

6.3.6. 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, the generated trace contains 364.7 million. The analysis of the 17 GB trace by our toolchain exhibits two functions with a high RDAM score: MultiWayMerge (RDAM of 0.83) and _write_nocancel (0.33). MultiWayMerge is a false positive and is discussed in Section 6.4.

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 has only a marginal effect on 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, we have verified that the I/O stack is a bottleneck with two experiments. We first measure with iostat that DC generates I/O disks at a rate of 178 MB/s, while the hdparm tool indicates that the disk maximum throughput is slightly lower: 162 MB/s (with a different workload, i.e., a sequential read, which explains why this maximum is lower). This first result also suggests that the I/O stack is overloaded. For the second experiment, we use a RAMFS partition to store the output file of the application. The resulting performance is naturally 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 RDAM score is caused by interference on the disk I/O stack, because the maximum RDAM score that we found by using a RAMFS is equal to 0.17.

6.4. False positives

Overall, we have found 3 false positives caused by parameter-dependent functions. We present in detail an analysis of these functions, and show that identifying these parameter-dependent functions is relatively easy.

Word_count In word_count, our toolchain 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).
PCA The loop of the PCA application summarized in Listing 5 has a high RDAM score of 0.37. We can easily show that this high score is a false positive. Each iteration of the instrumented loop mainly consists of another loop with (num_rows - i) iterations. Since the number of instructions in each occurrence of this loop is uneven, the score reflects this variation. When we instrument the inner loop, we find a score of 0.03 which reflects the real steadiness of the loop.

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.

6.5. Assessments

With this study, we show that the RDAM score can be used to identify lock contention, parallelism bottleneck, false sharing, NUMA placement bottleneck, I/O contention and network contention. Our study also shows that we can easily remove some of the bottlenecks by modifying few lines of code. Finally, we found 3 false-positives and we show that they were relatively easy to identify.

7. Related work

In this Section, we review existing works that aim at detecting interactions between threads and their impact on application performance. Several studies target the optimization of multi-threaded applications in general. In their work, Curtisinger et al. present Coz, a casual profiler [11]. 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. This work is complementary to ours. We identify, thanks to the RDAM metric, the set of sequences of instructions that suffer interference the most, but it does not prove that optimizing them will lead to better performance (e.g., if the sequence is not on the critical path). Coz identifies the sequences of code that contribute the most to the completion time, even if they do not suffer interference or do not have any performance defect.

Joukov et al. proposes OSprof, a work that focuses on measuring the completion time of function calls [24]. Joukob et al. propose to compare the “profile” of a function between several runs of the application in different settings (e.g. running one process vs running many concurrent processes). The functions profiles that differ significantly from one run to another are then reported to the developer for further investigation. As modifying the setting of an application also change the code executed by the functions, we consider that the RDAM metric can give more result, because we use the history of a single run with a single setting to identify interference bottlenecks. Moreover, while the RDAM score is able to classify different interference bottlenecks and to report the most problematic ones to the developer, OSprof cannot identify which of the identified bottlenecks hampers the performance the most.

Perfume [35] analyzes application logs and uses inference models in order to pinpoints performance bottlenecks. Perfume mines temporal properties from the logs, and pinpoints executions that violate inferred performance invariants (e.g., lower bounds, upper bounds, average). Our approach is complementary: we use the fastest execution as a reference to identify the time spent because of interference. Stitch [51] 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 performance bottleneck. Our work is complementary and rather focuses on identifying why one of the component is slow.

Apart from these studies, most of the other research projects focus on detecting one type of contention: memory contention (cache [10, 36, 48, 32, 45, 14, 15], NUMA effects [12, 25, 26], false-sharing [44, 22, 19, 50, 20, 29, 28, 34, 23, 27]), contention on the network [5, 9], or contention on locks [43, 49, 13, 4]. Overall, all these profiling tools and techniques only focus on interference on a single component of the memory hierarchy or on another hardware component, while we propose to simultaneously classify the performance impact of different interference problems. The RDAM metric does, however, not replace these tools and is complementary, as they are still required to understand why a sequence of instruction pinpointed by our profiling toolchain suffers interference.

8. Conclusion

Understanding the performance of a multi-threaded application is difficult because of the complex interactions between threads. In this paper, we propose the RDAM metric to quantify performance variation, and we show that it highlights interference, regardless of the interference cause. Our experiment with micro benchmarks and applications shows that our toolchain successfully detects contention with few false positives that are easy to discard.

Availability

The profiling toolchain, the experiments and the results are freely available at hidden-for-double-blind.
References


[18] Sanjay Ghemawat and Jeff Dean. LevelDB. URL: http://leveldb.org,


[52] Mingqiang Zhuang and Brian Aker. memaslap: Load testing and benchmarking a server.