THÈSE DE DOCTORAT

présentée à

L'UNIVERSITÉ PARIS-SACLAY

Spécialité

Informatique

soutenue par

Mohamed Said MOSLI BOUKSIAA

Date

PERFORMANCE VARIATION CONSIDERED HELPFUL

CONTENTS

1. INTRODUCTION

Computers are increasingly spreading in various areas which affect our daily life. Processing units are being embedded in devices that we use very frequently such as phones and other smart objects that will be soon present in almost every household. However, the impact of computers goes beyond the personal devices. We can think of numerical weather prediction, countless services available on the cloud, web applications running in huge datacenters, etc. Areas such as media processing, computational finance, and animation physics have been evolving in a way that requires more computing power and new techniques to efficiently and effectively process, and thus leverage, tremendous amounts of available data. Computer vision which is being increasingly used in video surveillance, character animation and computer interfaces is such a field [1].

Improving the performance of a computer had for a long time been straightforward: the more transistors we cram in the processing unit, the more speed we get. As predicted by Gordon Moore in 1965, this technique allowed the computer industry to double the performance every 18 months. Until recently, this allowed the developer of an application to effortlessly enhance its performance by upgrading the hardware. Towards the early 2000s, it became clear that this technique had its limits. In fact, below a certain size, transistors are unlikely to operate reliably and dissipating the energy that they use becomes harder at such a small scale.

From that point on, the most promising way to sustain Moore's law has been to focus more and more on parallel architectures which increase the core count instead of increasing the single core frequency. Fortunately, the arrival of chip-multiprocessors (CMPs) with ever increasing number of cores has made parallel machines ubiquitous. With this inevitable turn, applications that require additional processing power need to be parallel in order to leverage the available resources. However, achieving performance on a multi-core is difficult. Threads regularly interfere, either implicitly when they access the same shared hardware resource, or explicitly when they synchronize. When they interfere, the threads slow each other down, which decreases the parallelism and drastically degrades performance.

Identifying thread interference is difficult because interference can have many causes. Interference can come from any synchronization between the threads, and from any saturated hardware component: a cache, a memory controller, a disk, a network card, etc.

Identifying thread interference is also difficult because interference often remains hidden to the developer. This is obviously the case for an implicit interference, since the interaction between the threads is not explicit in the code, but also for an explicit interference, since any synchronization can silently become a bottleneck when the workload, the setting or the machine changes.

As interference often remains hidden, the developer needs profiling tools to identify the blocks of code that suffer interference. In order to be useful, a profiling tool should answer three different questions. First, the tool should identify *where* a code suffers interference (which line, basic block, function, or even component in a distributed system [2, 3], etc.). This information is required to know where the developer should optimize the code. Then, the tool should explain *why* a code suffers interference (contention on a cache, a network, a memory controller, etc.). This information is required to know how the developer should optimize the code. Finally, the tool should assess *how much* interference degrades performance. Since optimizing a multi-threaded application is long and difficult, this information is required to avoid wasting time on useless optimizations.

Unfortunately, current profiling tools are ill-suited to identify how much interference impacts performance. Some tools identify where and why the code potentially suffers interference by focusing on a specific interference cause [4–16]. These *cause-oriented* tools report incomparable metrics related to the cause (e.g., number of cache misses, I/O bandwidth), and are most of the time unable to assess how much interference impacts performance. Other tools identify where the code should be optimized in order to achieve better performance [17, 18]. These *where-oriented* tools are not designed to identify if the code suffers interference, and let alone to identify how much interference impacts performance. Other tools identify the root cause of a performance defect (where) by comparing the execution trace with a representative set of good and bad workloads [19]. These tools explain why a bad workload is inefficient, but they cannot identify interference hidden in both good and bad workloads, which also makes them inadequate to identify how much interference impacts performance in general.

As a result, today, in order to optimize a code, the developer often uses cause-oriented tools to identify why and where the code suffers interference. Since the developer often remains unable to identify how much a reported interference impacts performance, the developer spends weeks [19] trying to remove a randomly chosen interference pinpointed by one of the tools, without even knowing if the interference is at the origin of a performance problem.

In this thesis, we propose a new profiling tool to identify how much interference impacts performance. Our tool does not require prior knowledge, such as the prior identification of good and bad workloads, and performs an analysis of an application with a single run. For that purpose, instead of trying to identify where, why and how much interference impacts performance with a single analysis, we propose to decouple

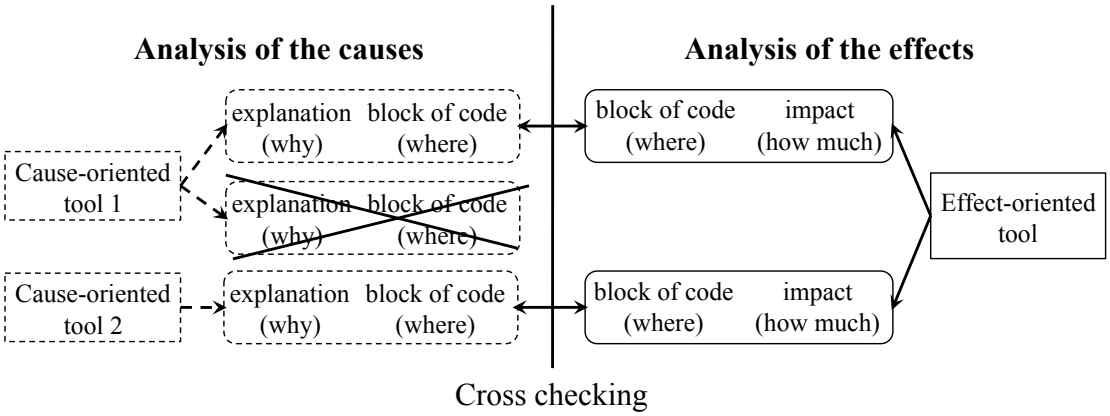


Figure 1: Cross checking of the causes and the effects

the analysis of the causes (where and why) from the analysis of the effects (where and how much). With this approach, as presented in Figure 1, the developer can cross-check the results of the two analyses in order to fully understand interference. The developer can identify all the blocks of code that suffer interference with the effect-oriented tool before trying to understand why with cause-oriented tools. The developer can also discard an interference bottleneck reported by a cause-oriented tool when the interference is not reported by the effect-oriented tool, since in this case, the interference does not degrade performance (e.g., the block of code in the middle in Figure 1).

Decoupling the analysis has two advantages. First, by eliminating the need to identify interference causes, we simplify the analysis of the effects. We do not have to try to understand why a block of code suffers interference: we only have to measure how much interference impacts its performance. Then, since the current cause-oriented tools are already able to efficiently identify interference causes, we can reuse them without any modification to perform the first analysis. Therefore, we only need a new tool that focuses on analyzing the interference effects.

Our idea to build an effect-oriented tool starts with a simple observation. While interference can have many different causes, it has only a single effect: interference slows the application down. This slowdown directly indicates how much interference impacts performance. Based on this observation, we propose a metric called RDAM (Relative DistAnce to Minimum) that captures this slowdown during a single run. We also propose an effect-oriented tool, called RDAMcalculator, that uses the RDAM metric to automatically identify both where and how much the code suffers interference.

We evaluate the usefulness of RDAMcalculator with 27 applications (7 from Splash2 [20], 7 from Phoenix-2 [21], 4 from Parsec [22], 7 from NAS Parallel Benchmarks [23], memcached [24] and LevelDB [25]). Our evaluation shows that RDAMcalculator can identify and classify different interference effects, regardless of the interference causes, with few and easily identifiable false positives. Thanks to RDAMcalculator, we identify

interference caused by false sharing, lock contention, poor parallelism, NUMA memory placement, network stack and disk I/O in real applications. In detail, we found that:

- RDAMcalculator is able to detect interference in 15 functions from 11 applications. Among the 15 functions, 3 (20%) are false positives. The remaining 12 functions pinpoint actual interference problems. 6 interference problems were previously identified in other works, while 6 are new.
- For the 12 true positives, RDAMcalculator successfully identifies where and how much interference impacts performance. By cross checking the results of RDAMcalculator with the results of classical cause-oriented tools, we show that we can easily explain where, why and how much interference impacts performance.
- Based on this analysis, we can easily correct 8 functions by modifying at most only 25 lines of code in each application, which leads to a performance improvement of up to 9 times.
- We show that the 3 false positives appear when a function seems to slow down when the workload varies, but not because of 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 thesis is organized as follows:

- Chapter 2 presents the general background of our topic. More specifically, this chapter starts by presenting parallel architectures and shared resources. In its second part, Chapter 2 presents the most frequent performance problems that occur as a direct consequence of accessing shared resources concurrently. The goal from this chapter is to get a sense of these phenomena that cause performance degradation in parallel architectures and why it is important to detect them in order to try and fix them.
- Chapter 3 presents the state-of-the-art tools that aim to detect performance problems and optimization opportunities. This chapter aims to reflect how different tools tackle the performance analysis goal from different perspectives. After an overview of the different approaches, we explain why an additional perspective is still needed to complete the existing work. More precisely, we conclude that we lack a tool which quantifies the impact of thread interference on an application while directly linking the identified interference effect to the affected code region in the application.
- Chapter 4 presents our work to bridge the gap identified in Chapter 3. We first present the RDAM metric that we rely on to detect occurrences of interference

between threads. We explain the intuition behind this metric and its formal definition. We then present RDAMcalculator, the tool we propose to automatically compute the RDAM metric for a given application.

- Chapter 5 presents an evaluation of RDAMcalculator with micro-benchmarks and with complete applications selected for their frequent use in validating new performance analysis techniques. This chapter provides evidence that RDAMcalculator successfully pinpoints interference effects regardless of their various causes.
- Finally, Chapter 6 concludes the thesis and discusses a few ideas to extend the presented work.

2. From architectures to performance bottlenecks

For a long time, manufacturers were able to leverage the shrinking of transistors to create processors twice as powerful every other year. This phenomenon was correctly predicted by Gordon Moore in 1975 and has been verified since then. However, at some point, the pace of advancement started to slow down and is expected to come to a halt soon. In fact, below a certain size, transistors will be unlikely to operate reliably. Moreover, dissipating the energy that they use becomes even harder at such a small scale (Power Wall). In search for additional ways to improve performance beside the single-core frequency, manufacturers relied on a variety of architectural designs that can leverage applications properties. Cache memories, which leverage temporal locality, as well as multi-core architectures, and SMPs, which leverage parallelism, are examples of such designs. Section 2.1 presents an overview of some of these features. Modern architectures are, however, more difficult to leverage. In fact, shared resources in these architectures are subject to contention which results in serializing the execution and has other subtle effects that degrade the application's performance in unexpected ways. In section 2.2, we explain some of these problems and show that they deeply affect the performance of the software that runs on such architectures and therefore need to be diagnosed efficiently.

2.1. Architectural solutions

2.1.1. More cores

The solution to the limitation faced by the manufacturers as to the single core frequency was to add more cores which allows to execute multiple execution flows at once. Multiprocessing is a generic term denoting the use of two or more central processing units (CPUs) within a single computer system. It regroups many different ways manufacturers imagined and realized to couple CPUs together. The different settings range from very tightly-coupled ones such as Symmetric Multi-Processing (SMP) multi-core architectures to loosely-coupled multiprocessors, typically clusters.

Figure 2.1, obtained using hwloc [26], shows a socket (to the left) comprising 4

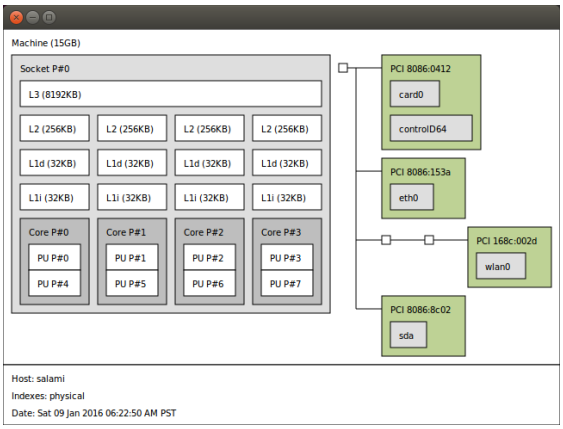


Figure 2.1: Multi-core architecture

cores, sharing memory and I/O resources (to the right). Coupling cores in this way can be considered as the building block for increasing core count. However, the number of cores that one can put on a single die sharing other resources is today roughly limited to 16. For this reason, more extensible architectures such as NUMA 2.1.3 appeared.

2.1.2. Cache memory

Cache memories came to solve a peculiar performance problem: the memory wall. This term denotes the disparity between CPU clock rates and off-chip memory rates, which basically means that memory is not keeping pace with the CPU in terms of speed. Therefore, an access to the Random Access Memory (RAM), which takes 50 to 150 nanoseconds, results in a waste of processing resources (several hundreds cycles where the CPU stalls). To soften the speed gap between the processor and the memory, there are local cache memories between the processor and the bus. The role of these faster, smaller memories is to leverage temporal locality of programs, by storing, closely to processing units, the data and instructions that are likely to be used shortly.

Cache memory levels In most multi-core systems, cache memories come in different speeds and sizes and can be either shared or private. That is why caches are categorized in levels:

- Level 1 (L1) cache is extremely fast (access time is typically 4 clock cycles) but relatively small (e.g. 64 KB).
- Level 2 (L2) cache is larger than L1 (e.g. 256 KB) and slower (a latency of around 10 clock cycles).

- Level 3 (L3) cache is both significantly slower (around 40 cycles) and larger than L1 or L2 (e.g. 4 MB to 40 MB), but is still much faster than the main memory.

Often, L1 and L2 are private while L3 is shared among the different cores as we can see in Figure 2.1.

Cache coherency Cache memories allow processors fast access to commonly used data, but requires to maintain consistency between the multiple copies of shared data. The most common coherence mechanism used to preserve consistency is based on invalidation where local copies are invalidated if a core updates a shared variable. For instance, in the MESI (Modified-Exclusive-Shared-Invalid) protocol, a memory block can be in one of the four following states:

- Modified: A block in this state is the only valid copy of the block. The memory does not hold valid information and no other cache may have a valid copy. The core that owns this block can write to it without notifying any other core.
- Exclusive: The first core to read in a block from memory will load it into the Exclusive state. This means that the memory has an up-to-date copy and there are no other cached copies in the system.
- Shared: As soon as a second core is reading the same block, it will be loaded to the cache of that core and will be marked Shared in all caches
- Invalid: As soon as one of the copies is modified by one of the cores, all other copies will be marked invalid and will need to be refreshed at the next access

Data is transferred between main memory and cache memory in blocks of fixed size (typically 64 bytes), called cache lines. Cache coherence mechanisms such as MESI are applied on a cache-line granularity. This means that even when only a small part of a cache line is modified, the whole 64-byte block is invalidated for the other threads which have a copy of the same cache line.

2.1.3. NUMA

Non-Uniform Memory Access machines allow to link multiple processors together in order to provide a single logical processing unit, therefore making it possible to scale to a much higher core count, which would not be possible exclusively with SMP (Symmetric Multi-Processing). A NUMA machine (Figure 2.2) is comprised of a set of nodes interacting via interconnect links, each node hosting a memory bank/controller and a limited number of cores sharing a single memory bus. The nodes are connected

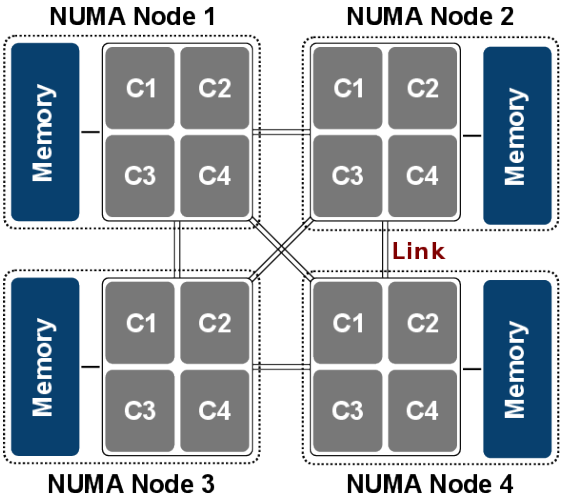


Figure 2.2: Simplified NUMA architecture

by means of a high speed cache-coherent interconnect (for example, QuickPath Interconnect (QPI) used by Intel or HyperTransport used by AMD).

Each core in a NUMA setting has access to memory banks in all nodes. Accessing memory on another NUMA node is called remote memory access, whereas accessing memory on the same NUMA node is called local memory access. A remote memory access is slower than a local memory access since the former has to go over the interconnect. In fact, even though the interconnect is fast, the signal path length from the processor to memory still has a significant impact. The exact cost differential between remote memory access and local memory access varies from an architecture to another and is commonly expressed as "NUMA factor", i.e. the ratio of a remote access' duration by a local access' duration. Different tested platforms showed NUMA factors ranging from 1.2 to 2.5 [27].

The cache coherence on a NUMA machine is usually maintained through what is called a directory protocol. In the directory protocol, memory blocks are still characterized by states similar to those presented in section 2.1.2. The directory, which is a memory structure, contains information about which processors have a shared cached copy of each memory block and which processor is the owner of the block. Explicit messages are then passed between processors to update a block's state when necessary.

2.2. PERFORMANCE PROBLEMS

The presented mechanisms allow to improve applications' performance. However, a misuse of these mechanisms can prevent developers from accomplishing a better performance, or even worse, degrade the application's performance. This section presents some of the features that lead to problematic situations performance-wise in multi-core

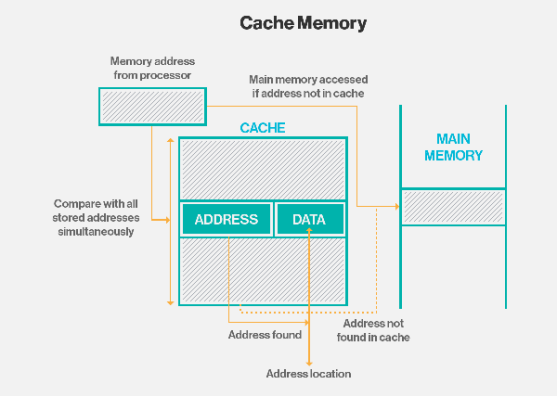


Figure 2.3: Cache memory

architectures and multi-threading.

2.2.1. Cache contention, false and true sharing

The use of last-level (shared) caches can improve performance by supporting on-chip inter-process communication and allowing heterogeneous allocation of cache to processes running on different cores. However, the existence of multiple copies of the same physical memory location at various levels of caches requires extra effort and sometimes costly operations to maintain a consistent view of the content of the memory. Therefore, the limited size of caches along with coherence mechanisms often lead to various contention problems.

When an application needs to access some data, it first looks for it in the cache (Figure 2.3). If the data cannot be found, it has to be fetched and loaded from a slower memory. This costly operation is called a cache miss. Cache misses can have different causes. For example, the first time that a data is referenced, an inevitable cache miss happens. Also, a thread can evict its own data from the cache when more prioritary data needs to be loaded and there is no more space left, so when the evicted data is needed again, a miss happens. However, in a multi-core context, cache misses can also be caused by interference between threads that share hardware and software resources. Cache contention problems happen in two kinds of situations. First, when different threads or processes which do not work on the same cache lines share a cache (cross-core interference), and second, when two threads work on a same cache line (sharing cache misses).

Cross-core interference When two threads or processes share the same cache, they may evict each other's data when they load their own. Whenever the owner of the evicted data needs it back, a cache miss happens. How often this happens depends on

whether both threads/processes are memory-intensive. As in this form of contention the two contending entities do not share any data, they can be separated. For this reason, a possible solution to this problem is to place threads wisely in order to balance memory-intensive threads with CPU-intensive ones and thus have less global demand on each shared cache memory. In other words, the solution is to opt for contention-aware scheduling [28].

Sharing cache misses Although a cache miss can happen at any cache memory level, here we address last-level cache misses since they are the most expensive. Sharing cache misses happen when two threads located on cores that do not share a last-level cache work on the same cache line, with at least one of them writing to it. After the first thread modifies the cache line, the copy held by the second thread is invalidated by the coherence protocol and needs to be updated in all cache levels that are not shared with the first thread before being reused. Since the coherence protocol operates at a cache-line granularity, we distinguish two cases.

True sharing In this case, the two threads are actually manipulating the same address in memory. Unless the application's logic permits the decentralization of the shared object, or the two threads can be assigned to cores sharing a last-level cache, sharing (and the cache misses that ensue) in this case is necessary to ensure the application's correctness.

False sharing Figure 2.4 shows two threads Thread 0 (blue) and Thread 1 (red) working on the same cache line. The colors indicate which part of the cache line is used by each thread. As shown in the figure, the actual variables being used by each thread are not the same, and yet at least one of the two threads is forced to load the whole cache line from the main memory each time it is invalidated. False sharing can be seriously harmful performance-wise as it can degrade application performance by as much as an order of magnitude [29]. Furthermore, false sharing is an implicit form of contention, often invisible in the source code, which makes it difficult to find. However, once detected this problem can often be easily fixed by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines (by adding paddings or utilizing thread-private variables).

2.2.2. NUMA: contention and memory placement

A thread running on Node 1, for example, has faster access to the memory in Node 1, than that of Node 2. The former is considered local memory, while the latter is remote, which calls for NUMA-aware strategies to avoid performance degradation due to unnecessary remote memory accesses. However, it has been observed that optimizing

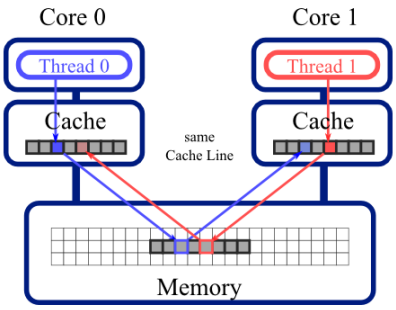


Figure 2.4: False sharing

memory placement for data locality alone does not solve all performance issues in a NUMA context. In fact, the high data traffic on the memory subsystem can be a serious bottleneck as well.

Bad locality Modern multicore systems are based on a Non-Uniform Memory Access (NUMA) design. To efficiently exploit such architectures, it is necessary to take the machine's physical layout into account. Particular attention has to be paid to remote memory accesses (i.e., main memory accesses performed from a core to a memory bank that is not directly attached to it). Remote memory accesses are a major source of inefficiency because they introduce additional latencies in the execution of instructions. These latencies are due to the extra hops required for the communication between a core and a remote memory controller (about 50% extra time). Therefore, the number of these accesses needs to be limited as much as possible as their impact on the performance of applications can be significant [28].

The presence of remote accesses can arise from various situations. A typical case is when the thread that allocates an object and the threads that access it the most are on different memory nodes. To give a simple but realistic example, let us consider a multi-threaded application where the main thread allocates and initializes an array before spawning N worker threads to do some processing on different parts of the array. Using first-touch (default Linux policy) as our memory allocation policy, the array will be allocated on the main thread's local memory, while worker threads will be spread across NUMA nodes, which results in a high rate of remote accesses. Another case is when several threads, on different nodes, need to access the same object. These situations can be fixed by carefully placing/migrating the threads, possibly coupled with NUMA-aware load balancing techniques. For example, ForestGOMP [30] is an OpenMP platform which enforces a distribution of threads that maximizes the proximity of threads belonging to the same parallel section. Other techniques rely on migrating data, or duplicating data when threads on different nodes need to access the same data simultaneously.

Contention on the interconnect and memory controllers Remote memory access has been a severe problem for a long time. In most recent architectures, however, another problem is taking over as a more impactful (though less frequent) impediment to performance, which is congestion on memory controllers and the interconnect. In fact, some applications can even perform significantly better with decreased data locality, when they succeed at mitigating data traffic congestion [31]. Because of congestion, memory access latencies can become as large as 1000 cycles, from a normal latency of only around 200. Such a dramatic increase in latencies can slow down data-intensive applications by more than a factor of three [31]. Solutions to this problem are built using the same basic techniques (migration and replication of memory pages), but with an important change in perspective. Instead of exclusively focusing on data locality, what needs to be optimized is global data traffic. In other words, strategies need to balance data traffic as evenly as possible between nodes.

2.2.3. I/O contention

In addition to performance problems that arise in computing phases, I/O operations can also lead to contention problems. Compute-intensive applications spend most of their time computing while I/O-intensive applications spend most of their time reading/writing to disk or communicating on the network. Detecting I/O bottlenecks for the latter is important to prevent severe performance degradation.

Storage disks Disk contention occurs when multiple processes try to access the same disk simultaneously. This problem is increasingly affecting applications performance in data centers. In fact, with the dramatic improvement in CPU speeds, the disk's maximal latency and throughput can become a bottleneck, which results in delayed responses and possibly request failures. Increasingly complex scientific applications, for example, require enormous computing power during the course of their execution, as well as huge storage space to store the checkpointing data generated for post-processing. As a side effect of the high degree of parallelism in such applications and the platforms they execute on, I/O contention at servers doesn't allow overall performance to scale with increasing number of processors [32]. Another study [33] has found that another type of applications (e-mail server), is characterized by its "bursty" workload, which means that peak I/O loads are significantly higher than the average load. If the storage subsystem is not provisioned for its peak load, its performance during peaks degrades significantly, resulting in I/O operations having significant latency. Large-scale web applications also suffer from disk's insufficient performance [34].

A variety of solutions have been proposed to mitigate this problem. One approach consists in delegating certain task types, such as file caching, consistency control, and collective I/O optimization to an exclusive small set of compute nodes, collectively

termed as I/O Delegate nodes [32]. To address the bursty workload problem, another technique [33] consists in allowing data written to an overloaded volume to be temporarily off-loaded into a short-term virtual store. The short-term store is created by opportunistically pooling underutilized storage resources either on a server or across servers within the data center. Writes are temporarily off-loaded from overloaded volumes to lightly loaded volumes, thereby reducing the I/O load on the former. While DRAM is frequently used as a cache to mitigate the performance gap between the disk and the CPU, RAMClouds [34] propose to completely rely on DRAM for the storage.

Networks Networks allow to connect multiple computers to have access to more computing power, or to link the client side of an application to its server side. Depending on network-to-node performance ratios, the raw performance of a network can be the system's bottleneck when the system's computational power outperforms its network capacity. This can even happen accidentally, as a defect can occur in the wiring due to human mistakes (we encountered this problem in the experiment described in section 5.2.5). On the other hand, applications that are deployed on a network have various aims and behaviors. Depending on their logic, applications can have very different computation/communication ratios, and thus put more or less stress on the network. How an application's performance is affected by the network is hence the outcome of both raw network performance and of the application's behavior. In addition to how an application interacts with the network, there is also one more side to the story, which is interference between different applications that share the same network.

Bad communication pattern The performance of an application that is deployed on a network depends on its communication-and-computation patterns. Some applications are "friendlier" than others network-wise. Applications with heavy communication needs, expensive all-to-all communications for instance, and/or whose design does not facilitate overlapping communication with computation (an example of such an application is FFTW [35], a Fast Fourier Transform library that uses hierarchical composition of multiple FFT algorithms, applied to perform a 2D transform of a 2000x2000 matrix) put much more stress on the network, which possibly becomes the system's bottleneck. On the other hand, applications (example: MCB [36], a Monte Carlo simulation code) which do not make use of global communications very often, and/or include intensive computation phases suffer less from the potential network/node speed disparity. Sometimes, one single application can alternate between computation-heavy phases and communication-heavy ones. An example of such an application is AMG [37], an implementation of the Algebraic Multi Grid Solver by using the Hypre library. Depending on the problematic situation at hand, the solution can vary from improving the network capacities (e.g. switching from Gigabit Ethernet (1 Gbps) to Infiniband (80 Gbps)) to an exhaustive redesign of the application. However, first of all, the developer should be

aware that the problem is originated in the network (or in how the application makes use of the network).

Interference from other applications Applications running on a network of computing nodes often share the network with other applications. For instance, large-scale HPC applications are usually submitted, as jobs, to supercomputers that they share with other jobs. Resource managers on these supercomputers use various allocation strategies, ranging from assigning a random set of nodes to more intelligent strategies providing better isolation for the job and better proximity between its processes. When an application runs on a "private" part of the network, its performance is not affected by other jobs sharing the network. This is true for the Blue Gene systems, unlike Cray XE6 systems [15]. Contention for links can cause significant performance degradation. This has been studied in [15], using pF3D, a diligently chosen parallel application (excellent computational and communication load balance in an ideal scenario). In this study, it is shown that the high variation observed in the performance of this application on a Cray system over months of runs is strongly correlated to inter-job interference. Experiments show that the character and location of other jobs running alongside of pF3D has a strong influence on the messaging rates (which in their turn directly affect the global performance of the application). During the different runs, various neighboring jobs were sharing the platform with pF3D, and affected it very differently. MILC [38], which is a communication-heavy application had a much higher impact on messaging rates than LSMS [39], which spends most of its time in computation and performs I/O at larger intervals than the duration of pF3D runs. All other parameters (job's size and shape) being identical between the two sets of runs, when the conflicting job is LSMS, the messaging rate was 27.8% faster than when the conflicting job was MILC. Multiple other cases gave strong evidence that the differences in performance are due to communication activities of competing jobs. Solutions for this kind of problem revolve around choosing a better resource management policy that minimizes traffic between disjoint sections of the network.

2.2.4. Thread synchronization: contention on locks

Above, we gave examples of contention that result from the sharing of hardware resources. Another type of contention that should be addressed happens because of shared software resources. Such contention is closely related to multi-threading, where multiple threads exist within the context of a single process such that they execute independently but share their process resources. Multi-threading is not the only way to write parallel programs, but there are reasons why it is widely used. For instance, in comparison to message-passing, multi-threading allows for an incremental parallelization of the software (e.g. using OpenMP) and is therefore less disruptive. Moreover, data structures

and control structures in many cases can be kept the same as in the sequential program in addition to saving memory by sharing data structures (unlike message passing).

Since threads share the same memory, synchronization mechanisms are needed to preserve memory consistency. Locks are a relevant example of such mechanisms that guarantees exclusive access to some shared data. It often happens that locks create contention on a multi-threaded application, which causes serialization. As a result, idling while waiting for a lock reduces parallelism and parallel efficiency [4]. Common solutions to this problem have been to design more efficient locking algorithms [40,41], or by adopting lock-free data structures [42].

2.3. CONCLUSION

The problems that hinder a parallel application's performance have various causes and are generally fixed using different techniques. By definition, a contention problem is linked to a shared resource that is being heavily accessed by multiple contenders. This chapter was an overview of the performance problems one can encounter when developing a parallel application, and the shared resources that are often contention sweet spots. Moreover, it served as a motivator as to why each of these problems should be detected, since they significantly hamper performance. Diagnosing performance bottlenecks in multithreaded applications will be of increasing interest as multithreaded applications spread to more fields and devices.

3. PROFILING TOOLS FOR MULTI-THREAD APPLICATIONS

As we could see from last chapter, interference between threads can take many different forms. To fix these performance problems, developers need profiling tools that pinpoint them. More specifically, to be able to solve these problems efficiently, a developer needs the answer to three questions. First, *where* a code suffers interference (which line, basic block, function, etc.). This information is required to know where the developer should optimize the code. Then, *why* a code suffers interference (contention on a cache, a network, a memory controller, etc.). This information is required to know how the developer should optimize the code. Finally, *how much* interference degrades performance. Since optimizing a multi-threaded application is long and difficult, this information is required to avoid wasting time on useless optimizations. In this chapter, we group the state-of-the-art tools according to which of these three questions they answer. The last section briefly introduces our contribution and explains how it completes the existing tools.

3.1. WHY: CAUSE-ORIENTED, PROBLEM-SPECIFIC TOOLS

This section presents the state of the art tools dedicated to the detection of a specific type of contention such as false sharing or NUMA contention. Whereas most of these tools can only say whether a given problem affects an application and pinpoint the affected code block, some of the more advanced can quantify the problem's impact on the application.

3.1.1. Cache-related problems

Cache-related performance issues has been a hot topic with many works addressing it from various angles. To diagnose cache problems in general, full cache simulation has been commonly used to obtain detailed cache behavior and start from there to detect cache contention problems. For instance CMP$im [8] is a cache simulator with many tunable parameters including allocation/replacement policies, write policies, number of levels in the cache hierarchy, etc. CMP$im gathers statistics such as the total number

of cache accesses and misses, sharing characteristics of multi-threaded applications, coherence traffic, etc. Some of the more advanced metrics reported by this tool are aimed to reflect whether the application would benefit from a shared cache or a private cache, and other insights that can help guide developers decisions. CacheIn [43] is a tool that uses simulation and monitoring to collect and report cache performance data including a false sharing detection algorithm which works by creating a serial trace of all memory references and comparing the address of shared writes to subsequent shared reads.

A major drawback common to this category of tools is the significant runtime overhead. By separately addressing specific cache-related issues, such as false sharing, or inter-process cache contention, some tools can significantly reduce this overhead.

Detecting false sharing To detect false sharing, the different tools rely on collecting and analyzing either memory accesses or cache-related events.

Instrumentation-based Since tools which provide a full simulation of cache memory events have overheads of 100x-plus, other approaches had to emerge. The category of tools presented in this paragraph instrument the application's code and insert extra instructions which perform a limited simulation of cache memory events. In fact, these tools sacrifice some of the accuracy to the benefit of efficiency, which turned out to be a good compromise.

We present two instrumentation-based tools that differ by their instrumentation technique (static, compiler-based vs. dynamic). Both tools have a significantly reduced overhead (5x) compared to full simulation and although both tools simulate only a part of cache events, they succeed at accurately identifying instances of false sharing.

Dynamic binary instrumentation and compiler-based instrumentation are two alternative approaches for performing instrumentation. They exhibit different tradeoffs of performance and generality. Dynamic binary instrumentors typically analyze the program's code just before execution in order to insert instrumentation. They introduce higher performance overhead, but the fact that they operate directly on binaries makes them extremely convenient. By contrast, compiler instrumentation inserts instrumentation in the compilation phase, which requires re-compilation of all source code, but provides higher flexibility.

Example of a dynamic-instrumentation-based tool: The authors in [10] reduce the tremendous overhead of full cache simulation by tracking cache events only when they are relevant to detecting contention that is due to sharing. More specifically, they recognize that cache misses that are due to a limitation in the size of cache memory (conflict and capacity misses) are irrelevant and can therefore be ignored. On the contrary, they track cache events that are produced by coherence mechanisms, namely cache line

invalidations and subsequent cache misses (which they call contention events).

Their tool returns a variety of interesting metrics, such as the total number of contention events and this same figure per instruction, which allows to pinpoint *delinquent access instructions*, that is to say the instructions that contribute the most to this type of contention. To achieve this, the tool keeps track of an ownership bitmap which shows which threads have accessed which cache lines. For every memory reference, this table is checked and updated such that the tool knows whether a cache invalidation/miss is going to happen and records it when applicable.

To address false sharing more specifically, authors extend the ownership bitmap to contain entries that record the access history for each word within a cache line. The occuring miss is then attributed to false sharing if the threads sharing the cache line do not actually share the same word within the cache line.

Example of a static, compiler-based tool: PREDATOR [13] also instruments the application before analyzing the collected data to detect false sharing. Instead of dynamic instrumentation such as in [10], PREDATOR uses an LLVM compiler phase to look for memory accesses and add instrumentation code to analyze them.

To identify false sharing, PREDATOR relies on a similar approach to that of [10] which consists in storing metadata for every piece of application data and keeping track of memory activity at a word level. A per-cache-line history table saves the properties (type: write/read, and thread id) of the two last accesses, which is sufficient to detect cache invalidations. The number of cache invalidations is the metric by which PREDATOR detects false sharing problems and ranks the severity of performance degradation due to them.

Once cache lines with many cache invalidations have been detected, PREDATOR needs to perform further analysis to differentiate actual false sharing from true sharing. To this end, PREDATOR tracks memory accesses per word, but once a word is accessed by multiple threads, it is no longer of interest since it is the object of true sharing. The per-word analysis also helps diagnose where actual false sharing occurs when there are multiple fields or multiple objects in the same cache line, which can greatly reduce the manual effort required to fix the false sharing problems.

The main novelty in PREDATOR compared to its predecessors is that it is able to generalize from a single execution to precisely predict false sharing that is latent in the current execution. There are two typical situations which can lead to the sudden appearance of previously-latent false sharing: when the cache line size or an object's starting address changes accross executions. In fact, those changes alter the data contained in each cache line and consequently the threads that access and share the cache line.

To be able to predict this latent form of false sharing, PREDATOR introduces the concept of a virtual cache line which is a contiguous memory range that spans one or more physical cache lines. A virtual cache line in the current execution represents a pos-

sible real cache line in either of the two aforementioned scenarios. PREDATOR looks within virtual cache lines for couples of memory words that are written by different threads.

OS-related approaches Although much faster than full simulation, the instrumentation-based tools we presented induce a prohibitive overhead that make their deployment often impractical. The category of tools we present in this paragraph implement system-level detection and mitigation/avoidance mechanisms for false sharing. These tools exhibit a reasonable overhead, which makes them much more suited for deployment. However, the reduced overhead often comes at a cost regarding portability.

Sheriff [44] is a functional replacement for the pthreads library which proposes to turn threads into processes. Sheriff simulates a shared address space using a shared memory region, replaces thread-related calls by their process counterparts (waitpid instead of pthread join for example) and uses inter-process mutexes to support POSIX synchronization operations. Sheriff provides two tools based on its "threads-as-processes" framework.

- SHERIFF-DETECT tracks write operations by different threads to a page to gather information about whether pages are shared or not. Page-protection allows Sheriff to track which threads access a page, and therefore whether multiple threads are trying to write to the same page. Whenever a page is shared, updates to this page are done locally by each process between synchronization points and diffs are committed whenever a synchronization primitive is encountered. SHERIFF-DETECT associates a per-cache line status with each cache line in every tracked page, which allows to identify interleaved writes (by different threads) to a cache line, which leads to cache line invalidations.

- SHERIFF-PROTECT uses the same mechanism for preventing false sharing altogether, eliminating the need for programmer intervention. This is of interest because it is sometimes difficult or impossible for programmers to remove the detected false sharing. For instance, padding data structures can cause excessive memory consumption or degrade cache utilization. Time constraints may prevent programmers from investing in other solutions, or the source code may simply be unavailable. The fact that Sheriff operates on shared memory pages privately and differs committing diffs to synchronization points eliminates repeated cache invalidations and thus avoids false sharing.

Sheriff reports false sharing problems accurately and precisely with reasonable overhead (around 20%). However, it can only detect write-write false sharing, and only works for programs that use the pthread library. It also fails to detect false sharing for

programs that use ad-hoc synchronizations or share data in the stack.

Plastic [45] solves two challenging problems. First, it is capable of rapid, low-overhead detection of false sharing in unmodified, running applications. Second, it resolves identified instances of false sharing using its own memory remapping facility.

To detect false sharing, Plastic relies on hardware performance counters to monitor coherence invalidation events, which indicate multiple cores competing for exclusive ownership of a cache line.

Whenever an abnormally large number of coherence invalidations is observed, an efficient multi-stage process is triggered to accurately identify the problem with minimal overhead. First, physical pages where contention is occurring are isolated, then memory accesses to these pages are sampled for short periods to find the accessed bytes in these pages and the identity of the accessors. This byte-level access log is parsed and contended cache lines are identified as the ones having multiple accessors with at least one writer. From this point forward, the task of Plastic is to transparently remap the contended regions to physical addresses on independent cache lines.

Since virtual-to-physical address remapping has a page granularity due to the MMU, Plastic uses dynamic instrumentation to remap addresses so that colocated objects are moved to separate cache lines. Plastic is currently implemented on the Xen virtualization platform, but the approach in itself is not specific to hypervisors: Plastic could be incorporated into an operating system. Nonetheless, Plastic's shortcoming is that the subpage memory remapping mechanism is not currently supported by most existing operating systems, reducing its generality. In addition, Plastic cannot pinpoint the exact source of false sharing.

PMU-based approaches PMU-based tools are introduced due to performance reasons. These tools rely on cache events collected from running programs. Performance monitoring units (PMUs) in processors can count many hardware events with low overhead and one could easily collect the desired counts via APIs such as PAPI [46] or tools such as perf [47]. This approach also provides high portability across operating systems and platforms.

A Machine Learning approach: Jayasena et al [48] take a peculiar approach to false sharing detection. Instead of trying to directly identify false sharing, they seek to derive its potential pattern. The use of machine learning comes from the insight that interesting information can be deduced from different kinds of performance event counts from running programs, but that such data are too overwhelming for human processing.

In this work, supervised learning is used to train a classifier with a set of sample kernels (mini-programs) - with and without false sharing. The trained classifier is then used to analyze memory access patterns of arbitrary programs. First, a candidate list of

performance counters is selected: for instance events that correspond to memory access (loads and stores), data caches (e.g., cache line state, cache misses), TLBs, interaction among processor cores, and resource stalls. The mini-programs are then used to compile a shorter list of events that are actually able to distinguish executions with false sharing from executions that are free from false sharing.

Performance overhead on programs in this approach is minimal. Program execution time often remains almost the same or insignificantly increased, at most by 2%, when collecting performance event counts. However, this approach cannot report all existing false sharing problems and cannot provide sufficient information for optimization.

Cheetah [14] brings three main improvements over the aforementioned tools. First, Cheetah does not require a custom OS, nor recompilation and changing of programs. Second, its false sharing detection technique is more efficient, with only 7% performance overhead. Instead of collecting and analyzing each and every memory access, Cheetah uses a PMU-based sampling to track only one memory access out of a predefined number of accesses. Even with sparse samples (e.g., one out of 64K instructions), Cheetah can identify false sharing with a significant performance impact. After capturing a memory access, Cheetah performs a series of analysis steps (locating problematic cache lines, computing cache invalidations, and reporting false sharing) that are much like PREDATOR's. Finally, it can precisely assess the performance improvement that would result from alleviating a false sharing problem without applying an actual fix, with less than 10% difference. Consequently, developers can avoid unnecessary manual effort leading to little or no performance improvement.

Data-centric approach Pesterev et al. [49] argue that costs due to frequent cache misses on a given piece of data may be spread over instructions throughout the application. Therefore, the approach used by typical profilers which attributes costs to specific code locations can overlook the significance of a cache miss problem. By reporting the data types (instead of code) with the most cache misses, Dprof helps programmers locate data structures that suffer misses in many places in the application's code. Dprof distinctly reports different kinds of cache misses: invalidation-induced, capacity misses, and conflict misses, which is important in order to select the right strategy to fix the problem. Dprof's reporting system consists in four different views ranging from the most generic one, a list of data types, sorted by how many misses they suffer to more detailed presentation of memory activity down to the associativity sets used by hot memory objects.

Summary A variety of tools and methods can be used to detect false sharing. A full simulation of cache memory is both inefficient (x100 overhead) and unnecessary if the sole purpose of the analysis is to detect false sharing. This insight was used by instrumentation-based tools [10, 13] which perform only a partial simulation of cache

events that are relevant to false sharing detection. Although these tools are effective, they still present a significant overhead (x5). Some other tools based their approaches on modifying the underlying OS [44, 45]. However such tools succeed at bringing the overhead down to a mere 20% of the application's execution time, their weakness is portability as they are only applicable to specific kinds of applications.

By relying on PMU-based sampling, Cheetah [14] both reduces the overhead to 7% and preserves the portability benefit of instrumentation-based tools. Orthogonal to these efforts, Dprof [49] reports cache misses suffered by each data object instead of attributing them to instructions. Knowing that the other tools generally use impact thresholds below which they do not report false sharing instances, Dprof can be useful to unveil false sharing that affects a piece of data in different locations in the code and whose total impact is significant.

Inter-process cache contention

CAER [50] is a Contention Aware Execution Runtime environment oriented towards data centers for web services, where applications with different needs co-exist. While some applications are more concerned with latency requirements (the latency-sensitive category), others have throughput requirements (the batch category). CAER first detects inter-process cache contention then responds to it.

CAER collects cache miss rates using hardware performance monitors and then feeds them to one of its two heuristics which run continuously to detect contention. Each of these approaches aims to supervise the interaction between neighboring (sharing cache memory) applications of opposite types.

The first approach, *burst shutter*, provokes sudden bursts of execution in the batch applications and assesses the resulting increase in cache miss rate of the latency-sensitive applications. The second approach, *rule based*, observes cache miss rates of the two kinds of applications until their average rises above a threshold. When a contention is detected, CAER responds using a fine grained throttling of the execution of the batch application to relieve pressure in the shared cache.

CAMP [51] is a performance model that estimates the performance degradation due to inter-process cache contention. To this end, a number of metrics are computed. Since this model deals with processes running concurrently and sharing a cache, it first evaluates the effective cache size of a process, which is the average number of cache lines occupied by the process in a set. The second needed metric is the cache line reuse distance, which is the number of distinct cache lines accessed between two consecutive accesses to the cache line. Combining these two figures allows to compute MPA (Miss Per Access) and then deduce SPI (Seconds Per Instruction). The prediction made has an average error of 1.57% on CMPs that have different cache sizes than the one where

the processes were initially profiled. This model thus helps guide process assignment decisions effectively.

AQL_Sched [52] tries to prevent cache contention, among other goals, instead of detecting it. AQL_Sched is a scheduler designed for cloud data centers where virtual machines from different users are colocated in the same physical machine. This tool characterizes applications dynamically according to how sensitive they are to sharing resources (CPU, memory, I/O, etc.) with other applications.

For instance, some applications (denoted as "Last-level cache friendly") are very sensitive to cache pollution. In contrast, some other applications are not affected by sharing of Last-level cache, but may or may not affect the performance of other applications. The type recognition system periodically collects metrics and computes the current category to which the workload belongs. According to the results of this categorization, different scheduling quantum values are allotted to different workload types, and clustering techniques are applied to avoid performance degradation.

Summary Inter-process cache contention happens when different applications share a cache memory and evict each other's cache lines in a way that degrades their performance. Every tool [50–52] dealing with this problem starts by assessing the impact of contention by measuring cache misses and deducing a chosen metric which generally has an associated threshold that indicates a problematic situation. Some tools [50, 52] go beyond assessing the impact and respond to the problematic situation online. Such tools use a classification of the running applications according to the sensitivity of their performance to cache-related problems. Based on this classification, clustering and/or scheduling strategies are employed to either fix the problem or prevent it from happening.

3.1.2. Detecting I/O contention

Detecting Disk contention Related work dealing with disk contention can be divided in two categories according to how the problem is perceived. Most approaches deal with cases where disk contention is a permanent problem, and there is no escaping it as long as we keep using the classic storage techniques [32, 34]. Rather than supervising the application while it is running, detecting a potential problem, and then applying a fix online, this category focuses on redesigning storage systems. For instance, fully relying on DRAM [34] and delegating I/O operations to a small set of nodes [32] have been proposed.

A few other approaches deal with the disk contention problem as an occasional phenomenon that is encountered from time to time or, more generally, as the result of factors that should be monitored dynamically such as the distribution of data popularity. These

approaches focus on detecting such unpredictable bursts, and dynamically coping with them. Everest [33] detects peak I/O request rates that are over an order of magnitude higher than average load and temporarily off-loads data written to an overloaded volume to lightly loaded volumes. To achieve this, Everest monitors the queue length (number of pending I/O requests). When this number is greater than 32, it utilizes spare bandwidth on other storage volumes to absorb writes performed to the overloaded volume. Ananthanarayanan et al [53] use the varying popularity of files as an indicator for contention in MapReduce clusters. In fact, the significant disparity in data's hotness in these clusters results in the more popular content becoming a bottleneck. To solve this problem, they propose Scarlett, a tool which replicates files based on predicted popularity. For every file, Scarlett maintains a count of the maximum number of concurrent accesses (which reflects the popularity) during the last 24 hours, and then computes a replication factor that is proportional to its popularity. Oh et al [54] deal with hybrid storage solutions which use Solid State Drives (SSDs) as non-volatile cache. This peculiar kind of cache requires an over-provisioned space (OPS) (used for garbage collection) in addition to the normal caching space. The authors aim to optimize the performance of this caching system by finding the optimal space fraction that should be allotted to OPS. To this end, they propose a dynamic scheme based on cost models for a given workload.

Detecting Network contention Bhatele et al [15] study performance variability in a parallel application (pF3D). The metric used to evaluate performance is the average throughput (MB/s) since an experiment shows that a high throughput leads to low execution time and vice-versa. The authors consider performance variability bad as it prevents developers from accurately assessing the impact of their optimization on the code. Therefore, the profiling done here aims to correlate the selected performance metric (average throughput) to the potential issue that causes its variability. They conclude that interference from other parallel applications sharing the same network links is the actual culprit.

Casas et al [16] provide two types of measurements to evaluate the relationship between network capability and application performance based on injecting extra packets. The first benchmark is concerned with how an application impacts the network. A low-overhead MPI ping-pong application is first run on an empty network then concurrently with the main application. The latency distribution of the extra traffic is measured in both cases. The difference between the two distributions is then used to infer the level of perturbation caused by the main application on the network.

The second benchmark aims to evaluate the performance an application would have on less capable networks or when it shares the network with other software components. The benchmark simulates reduced network capability by aggressively injecting network traffic while the application runs. The extra workload is an MPI application whose processes form a ring and exchange messages at a customizable rate. Varying

the intensity of the extra workload allows to know how the application would respond to different degrees of interference. Finally, the different measurements are combined to make quantitative prediction of the performance degradation suffered by an application A when sharing the network with an application B. The first benchmark is used to quantify the impact of application B on the network, then the data previously produced by the second benchmark allows to predict the response of application A to such a level of perturbation.

Summary Network contention is detected using a variety of metrics. These metrics range from the simplest [15] (throughput) to more sophisticated ones based on latency distributions of extra traffic [16] intended for monitoring the network's activity. These more sophisticated metrics accurately capture both the disturbance that an application causes on the network and how an application responds to such a disturbance.

3.1.3. Detecting NUMA problems

MemProf [55] is a profiler which aims to help developers select an efficient technique to reduce the amount of remote memory accesses in their applications. Previous profilers do not track enough information to precisely pinpoint memory objects accessed remotely, neither can they identify opportunities for optimizing such accesses. To bridge this gap, MemProf keeps track of a memory object's history (which threads access which objects at any point in time during the run of an application) using instruction-based sampling. This history includes, for each access, properties such as the node from which the access is performed, the memory node that is accessed, the latency of the memory access, whether the access is a read or a write operation, etc.

MemProf provides a C API to process the history, which allows to compute statistics about a single thread or object, or about a group of threads or objects. Writing scripts based on this API allows the developer to understand the cause of a high remote memory access rate and therefore select the right solution. For example, in the case of a facial recognition application, MemProf showed that a single matrix out of 200 is responsible for most remote accesses and that it is written only once and then accessed in read-only mode by a set of threads. The problem was then fixed by duplicating this matrix on all nodes after its initialization, which resulted in a performance improvements of up to 41%. MemProf also comes with a set of generic scripts whose output is often sufficient for understanding the symptoms of an application, e.g., ratio and number of remote memory accesses, list of the most accessed object types, access patterns corresponding to an object type or to a specific object instance.

Carrefour [31] targets applications that generate substantial memory traffic, in which case congestion on memory controllers and interconnects can be a bigger issue than re-

mote memory accesses. Carrefour analyzes the memory accesses of applications from which it then deduces memory management decisions such as moving memory pages, where to allocate data, etc. In order to take sound memory management decisions for the aforementioned category of applications, Carrefour uses a three-step algorithm: measurement, global decisions and page-local decisions. The first metric to be monitored is the application's MAPTU (Memory (DRAM) accesses per time unit (microsecond)). Carrefour is enabled for applications with the MAPTU above a certain threshold, 50 being a reasonable value. Below this threshold the application's performance will likely not benefit from any memory placement strategy that is why Carrefour is shut down to avoid any useless overhead. A few other metrics serve to detect whether applying a particular memory management technique would take the application to a better or worse state contention-wise. For instance:

- MC-IMB (Memory controller imbalance): is the standard deviation of the load across all memory controllers, expressed as percent of the mean. This metric serves to check whether load on controllers is not evenly balanced (MC-IMB > 35%), and when it is the case, page interleaving is resorted to.
- MRR (Memory read ratio. Fraction of DRAM accesses that are reads) is used to decide whether page replication would benefit the application. More specifically, an MRR of 95% or more ensures that synchronizing pages across nodes will not be frequent enough to outweigh the benefits of replication.

Summary NUMA contention has been dealt with in the form of two major problems: 1) high remote access rate, and 2) congestion on the interconnect and memory controllers. In the first case, the profiling challenge is to report enough information to the developers in order to guide them to the root cause [55], in the code, of a high remote access rate. In the second case, the focus is put on metrics that reflect whether the system is in a global state of balance [31] and whether a particular memory placement strategy would alleviate a potential imbalance problem.

3.1.4. Detecting lock contention

Free Lunch [6] is a lock profiler for server-class applications (databases, web servers) which introduces a new metric for lock contention, called critical section pressure (CSP). CSP aims to indicate precisely whether thread progress is being impeded by a lock. This goal can not be reached using metrics centered around total critical section time or the number of acquisition failures. In fact, server-class applications run for long periods of time during which the demand put on locks can vary significantly. For this purpose, Free Lunch was designed for in-vivo use so as to measure CSP continuously over windows of one second.

CSP is the ratio of i) the time spent by the threads in acquiring the lock and ii) the cumulated running time of these threads. Therefore, CSP indicates the percentage of time where threads are unable to make progress. Free Lunch reports the identity of the locks whose CSP reaches a threshold. Free Lunch locates a previously unreported phase with a high CSP in the log replay subsystem of Cassandra. This issue is triggered under a specific scenario and only during a phase of the run.

Tallent et al [4] evaluate three different strategies for gaining insight into performance losses due to lock contention. Based on sampling, all three strategies start by determining whether a sample occurs during a work phase or an idleness phase to measure the amount of idleness suffered by the application. The strategies differ by the entity on which they "blame" the measured idleness. For example, the first strategy assigns an idleness sample to the calling context in which the waiting sample occurs. The second strategy accounts for an idleness sample evenly among the threads holding any lock when the sample happens. The authors show, however, that these two approaches are ineffective when it comes to complex applications, since they only help pinpoint symptoms (first strategy) or suspects (second strategy) at best.

That is why they propose their third approach in which idleness is accounted for in a per-lock counter, each time that a sample occurs in a thread waiting for that particular lock. Then, when a thread that possesses a lock releases it, the thread blames itself for all of the idleness (attributes that idleness to the context of its lock release operation) that accumulated while it held the lock.

Compared to previous approaches, Yu et al [5] argue that lock analysis done in isolation can overlook subtle interactions between an application's components and therefore miss interesting optimization opportunities. As a more holistic approach, they study cost propagation through both lock contention (delay propagation to the components waiting for the same lock), and call dependency (accumulated costs from callees to callers).

These two phenomena can combine to exacerbate contention through a snowball effect. An illustrative real-world case is presented in which three browser threads contend for access to a critical file operation section, whose lock holder called and is waiting for a device driver that is also contending for a lock with an AntiVirus thread and a configuration manager, etc. To accurately account for multi-layered contention problems such as the one presented, the authors use distinct metrics for total waiting time on the one hand and waiting time suffered due to cost propagation on the other hand. Comparing the two classes of metrics provides an upper bound for the optimization of cost propagation.

Summary Detecting lock contention by only measuring the time spent waiting for a lock has proved insufficient. Some works propose more relevant metrics [6] to detect contended locks that are worth fixing and that would not be detected using basic metrics. Others focused their effort on detecting which thread is to blame [4] for the time wasted

	why	where	metric reflecting loss
Zhao et al [10]	false sharing	instruction	(# misses + # invalidations) / # instructions
PREDATOR [13]	false sharing	line in the source code	# invalidations
Sheriff [44]	false sharing	memory object	# invalidations
Plastic [45]	false sharing	Memory word	-
Jayasena [48] et al	false sharing	-	-
Cheetah [14]	false sharing	object address + code line	speedup if fixed
Dprof [49]	false sharing	data types	# cache misses
CAER [50]	cache contention	-	# cache misses
CAMP [51]	cache contention	-	seconds per inst.
AQL_Sched [52]	cache contention	-	-
Everest [33]	disk contention	overloaded volume	-
Ananthanarayanan et al [53]	disk contention	popular file	-
Oh et al [54]	disk contention	-	GC cost
Bhatele et al [15]	network contention	net. links used in //	-
Casas et al [16]	network contention	-	Δ packet latency
MemProf [55]	NUMA placement	Memory object	# remote accesses
Carrefour [31]	NUMA congestion	-	-
Free Lunch [6]	lock contention	contended lock	% impeded progress time
Tallent et al [4]	lock contention	lock + responsible thread	idle time
Yu et al [5]	lock contention	device drivers	propagated delay

Table 3.1: Summary of cause-oriented tools

on waiting for locks. Finally, some other works focus on the necessity of not studying the lock contention problem in isolation [5] and study the cost propagation of waiting for locks across the execution path.

3.1.5. Conclusion

Table 3.1 summarizes the different cause-oriented tools presented in this section. The *why* column indicates the problem type (the cause of performance degradation) addressed by the tool. The *where* column indicates the contended entity reported by the tool. When a "-" is indicated, this means that the tool estimates in general whether the resource suffers a problem or not (application having false sharing, a congested NUMA system, a congested network, etc.). The last column indicates whether the tool provides a metric that evaluates performance degradation due to interference. Where there is a "-", the tool is rather prediction- or prevention-oriented. In these cases, actions are taken to prevent a performance loss from happening. Cells of this column are highlighted in

green when the corresponding tool provides a direct information on the global impact the detected problem has on the application's performance.

In sum, the questions these tools answer well is *why* and *where*. Although some of them can give insight about the *how much*, they are only helpful for the specific problem they are specialized in. Since the metrics computed by these tools are incomparable, their mutual results cannot be used together to get a global understanding of what affects the application's performance the most.

3.2. WHERE: GENERIC, CAUSE-OBLIVIOUS TOOLS

COZ [18] uses causal profiling to indicate where exactly programmers should focus their optimization efforts and quantify their potential impact. To predict the usefulness of reducing the execution time of a particular line of code without having to actually do it, COZ uses virtual speedup to mimic the effect of optimizing a specific line of code by a fixed amount. An instruction is virtually sped up by inserting pauses to slow all other threads each time the line runs. Virtual speedup is varied from between 0% (no change) and 100% (the line is completely eliminated) in order to predict the effect of any potential optimization on a program's performance. COZ uses throughput as its performance metric.

To profile throughput, developers specify a progress point, indicating a line in the code that corresponds to the end of a unit of work. COZ's profiler thread then randomly selects a line to virtually speed up, and a speedup factor. Then the profiler thread saves the number of visits to the progress point. COZ then uses sampling, and every time a sample is available, a thread checks whether the sample falls in the line of code selected for virtual speedup. If so, it forces other threads to pause. This process, called an experiment, continues for a pre-defined time. After a short pause, a new experiment is started to collect more figures. The results are combined at the end to produce profile graphs for the lines of code that were virtually sped-up. After that, it is up to the user to interpret them and make an educated choice about which lines may be possible to optimize.

OSprof [56] is designed to be a versatile, portable, and efficient OS profiling method. For this purpose, metrics that are too specific or costly to collect are avoided and the approach is simply based on the analysis of latency distributions of OS operations (mostly system calls). Graphic distributions are generated and then examined by the user to draw meaningful conclusions using the user's knowledge of the characteristic times of the used architecture (duration of a context switch, the scheduling quantum, etc.). The analysis is based on considering that each peak in the distribution corresponds to a different execution path.

To further assist users in their analysis of the profiles, OSprof comes with automatic processing and visualization scripts to present the results clearly and concisely. For in-

stance, some of these tools report how two distributions differ in terms of the number of peaks and their locations. Using OSprof, users can for instance discover the existence of multiple execution paths (only some of which having to wait for a lock for example) for the same operation. Also, comparing latency distributions of two runs with different parameters can be helpful. For example, a sequential run with a one-latency peak compared to a parallel run with two latency peaks shows a contention between processes in the parallel execution scenario.

Song et al [19] propose to evaluate the effectiveness of statistical debugging for detecting performance problems. Statistical debugging has been successfully used to detect functional bugs by collecting program predicates (branches, function returns, etc.) during both success runs and failure runs, and then using statistical models to automatically identify predicates that are most correlated with a failure. To adapt statistical debugging to performance problems, Song et al first had to define what good inputs and bad inputs are. The second challenge was the design of predicates and statistical models that are suitable for the problem under diagnosis.

The first challenge was solved by using performance-bug reports filed by users which provide two sets of inputs (good/bad runs). For the second challenge, they select pertinent predicates (potentially revealing metrics) and statistical models to process those predicates. The chosen predicates in this work are: branches (whether they are taken or not), functions returns values, and scalar pairs (for each pair of variables, whether they are equal, or the first is greater than the second, etc.) As for the statistical models, two models were evaluated: the first one checks the presence (at least once) of a predicate in a run, while the second bases its assessment on the exact number of times the predicate has been true in a run.

The authors conclude that the use of branch predicates under both statistical models provides almost full coverage of the 65 studied performance problems. However, they also found that although useful, statistical debugging can almost always provide useful information for performance diagnosis, developers still need help to figure out the final patches. Especially, when an inefficient loop is pointed out by the statistical model, developers need more program analysis to understand why the loop is inefficient and how to optimize it.

Summary Table 3.2 summarizes the where-oriented tools we presented in this section. Most where-oriented tools help pinpoint locations in the code that are theoretically worth optimizing [18, 56]. However these tools provide insightful information, they do not tell us whether the indicated locations actually hide a performance bug. A significant manual effort is still required from the developer to only check the presence of a real performance problem. Such tools are not designed to identify if the code suffers interference, let alone to identify how much interference impacts performance.

Some other tools that take interest in the *where* question identify the root cause of a

	why	where	metric reflecting loss
COZ [18]	-	code line	resulting speedup = f(% optimization of line)
Song et al [19]	-	predicates correlated with failure	- (provided by user)
OSprof [56]	-	a system call	-

Table 3.2: Summary of where-oriented tools

performance defect by comparing the execution trace with a representative set of good and bad workloads [19]. These tools explain why a bad workload is inefficient, but they cannot identify interference hidden in both good and bad workloads, which also makes them inadequate to identify how much interference impacts performance in general. Also, the effectiveness of this category of tools relies on the availability of pertinent workloads provided by the application's users.

3.3. ANOTHER COMBINATION: WHY AND HOW MUCH

A complementary approach to analyze thread interference is also proposed by Eyerman et. al [57]. In their work, they propose to measure, with hardware counters, 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 interference.

3.4. CONCLUSION

In this chapter, we reviewed state-of-the-art tools that are intended to help developers fix performance problems in their applications. The various tools tackle performance diagnosis from different angles. Some of the tools allow to answer *why* and *where* performance is degraded. Other tools answer the *where* question in a generic way, and finally, some other works tackle the combination *why* and *how much* performance is degraded.

For the time being, there is no tool answering the question pair (*where*, *how much*), thus allowing the developer to quantify performance loss due to interference while associating it to a location in the application. To reach a complete understanding of a performance problem, such a tool is needed to generically pinpoint occurrences of interference in the application before delving into a thorough analysis of the causes and possible solutions using *where*-oriented tools. For this purpose, we present in the next chapter a new generic tool that aims to be a point of connection between the existing tools.

4. RDAM AND RDAMCALCULATOR

As we have seen in the previous chapter, we need a new tool that focuses on analyzing the interference effects. For this purpose, we propose to assess the time wasted due to an interference, or put otherwise, the time we would gain by fixing an interference. To quantify this effect, we propose the RDAM metric which we present in section 4.1. In section 4.2, we present a tool which computes the RDAM metric for a given application.

4.1. THE RDAM METRIC

The RDAM metric aims at identifying the effects and locations of the interference instances a multi-threaded application suffers. This metric identifies (i) where the code suffers interference (which function(s) or loop(s)) and (ii) how much interference impacts performance.

4.1.1. Interference results in slowdown

In order to define the RDAM metric, we start with our simple observation: interference slows the execution down. Therefore, we want the RDAM metric to reflect this slowdown. However, capturing the slowdown caused by interference is difficult because we cannot easily estimate 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 [19, 56]. This solution is not satisfying because changing the setting often drastically changes the executed code. For example, with OpenMP, varying the number of threads used to execute a loop in parallel changes the number of iterations executed by each thread, which makes the comparison between the runs difficult.

Performance variation as an indicator In order to compute the slowdown caused by interference, we rely on an intuition: performance variation is the universal indicator of any interference issue. More precisely, if a block of code suffers interference, sometimes, it will execute slowly because other threads use the same hardware resources

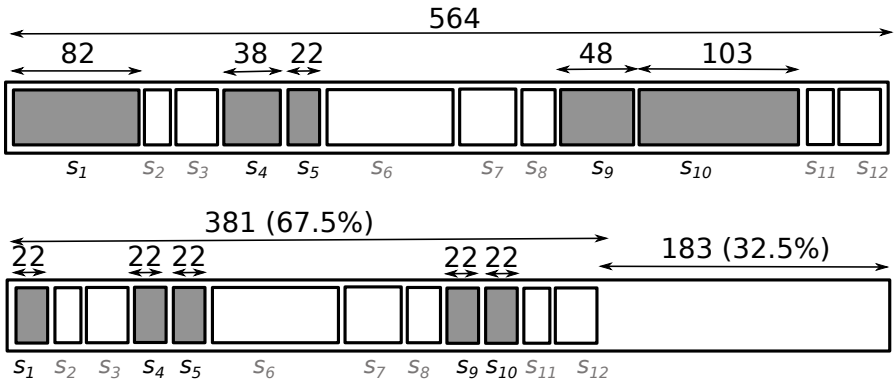


Figure 4.1: Illustration of the RDAM metric: based on the original execution trace (on the top), an ideal improvement without interference is estimated at 32.5% (on the bottom)

or delay a synchronization, while sometimes, it will execute quickly because the other threads do not interfere.

Based on this hypothesis, we can assume that, if a block of code is executed often, it will probably also execute at least once with little interference. We can thus consider that the fastest execution is almost interference-free, and that any slower execution is caused by interference. We can then compute, for any block of code, the slowdown of a thread caused by interference.

We define this metric as the RDAM score (Relative DistAnce to Minimum) of a block of code. Figure 4.1 illustrates the principle. Each box represents a sequence of instructions executed by a thread, and the gray boxes are repetitive sequences of instructions. If we assume that the sequence with the minimum duration (in this case, 22) is only slightly slowed down by interference, the RDAM score approximates the performance improvement (represented at the bottom) that would have been obtained if each occurrence of the sequence had been executed without interference.

Finally, it is worth noting that by using the fastest execution as a reference, RDAM is different than average-execution-centered metrics in that it can detect that an application suffers interference even when most occurrences are slowed down.

4.1.2. Formal definition

If s_i is a sequence of instructions, we say that s_i and s_j are repetitive sequences of instructions, and we note $s_j \sim s_i$, if they contain the same instructions. Then, we define R_i as a set of repetitive sequences of instructions: $R_i = \{s_j | s_j \sim s_i\}$. In Figure 4.1, the set of gray boxes form an R_i: $R_1 = \{s_1, s_4, s_5, s_9, s_{10}\}$. Finally, if we note d_i the execution time of s_i and $\overline{d_i}$ the minimal execution time of the $s_j \in R_i$ ($\overline{d_i} = min_{j|s_j \in R_i}\{d_j\}$), our hypothesis says that $\overline{d_i}$ is only slightly slowed down by interference. If T is the

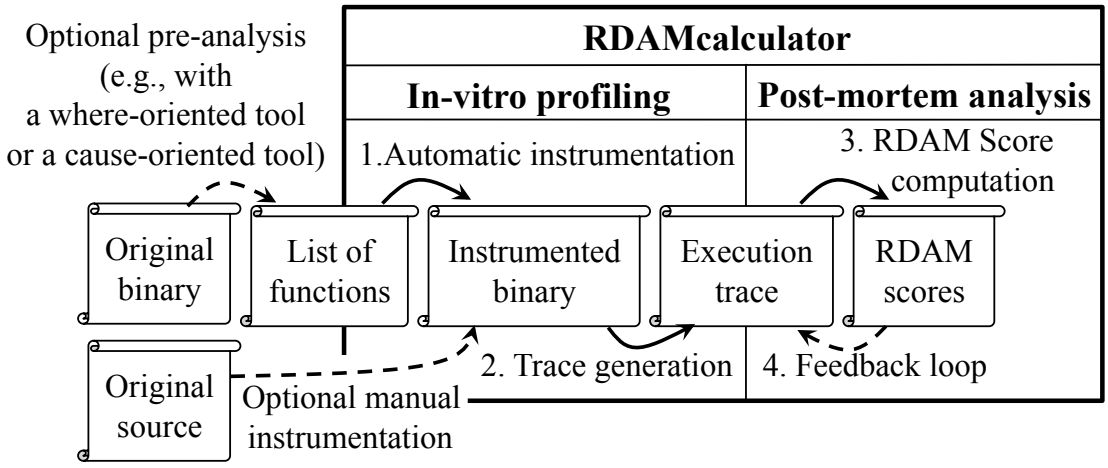


Figure 4.2: Overview of RDAMcalculator.

execution time of a thread, we can then approximate the slowdown of the thread caused by interference when we execute the $s_j \in R_i$ with:

$$RDAM_i = \frac{\sum_{j|s_j \in R_i}(d_j - \overline{d_i})}{T}$$

From this formula, it results that $RDAM_i$ is a number between 0 and 1. A higher $RDAM_i$ indicates that a thread suffers more interference. Being directly associated with a sequence of instructions s_i, in addition to indicating an interference phenomenon, $RDAM_i$ pinpoints the code block where the interference happens.

4.1.3. Accuracy of the RDAM metric

In our equation, we consider that performance variation can only come 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 RDAM score.

Moreover, in our equation, we consider that $\overline{d_i}$ is almost an interference-free execution. In practice, we cannot prove that this fastest execution does not suffer interference. However, since we record thousands to millions of occurrences, the probability of capturing an execution with little interference should be large. We confirm this hypothesis in Section 5.1 by experimentally showing that $\overline{d_i}$ is close to the time taken by an interference-free occurrence in four different micro-benchmarks, even in highly contended cases.

4.1.4. Conclusion

In order to detect interference, we use the RDAM metric which reflects the slowdown suffered by a thread. To approximate this slowdown, we compute the RDAM score for highly repetitive sequences of instructions assuming the presence of an almost interference-free occurrence. In the next section, we present a tool which allows to compute the RDAM metric.

4.2. THE EFFECT-ORIENTED PROFILING TOOL

RDAMcalculator is an effect-oriented profiling tool. It analyzes the interference effects on an application by automatically computing the RDAM scores. As presented in Figure 4.2, RDAMcalculator splits the analysis in two phases: an in-vitro profiling phase and a post-mortem analysis phase.

During the in-vitro profiling phase, RDAMcalculator automatically instruments a binary in order to add probes that record timestamps during the execution (step 1). RDAMcalculator then runs the instrumented application and records the timestamps in an execution trace (step 2). During the post-mortem analysis phase, RDAMcalculator computes the RDAM scores by analyzing this trace (step 3).

In this section, we discuss where RDAMcalculator adds the probe in the code, we discuss why RDAMcalculator can report false positives, and we finally describe in detail the different steps of RDAMcalculator.

4.2.1. Selecting the locations of the probes

Selecting the locations of the probes is challenging. If RDAMcalculator inserts too many probes, the execution time of the probes drastically changes the timing behavior of the application, which makes the RDAM scores inaccurate. We have measured that the execution of a single probe that only reads the CPU cycle counter costs 40ns (88 cycles) on our large machine (Opteron48, see Section 5.1).[1] As a consequence, adding a probe at the beginning and at the end of each function already leads to a very large overhead for many applications.

Because of the overhead, RDAMcalculator can only insert few probes, and in carefully chosen locations. In our experiment, we chose to capture, by default, the time taken by a list of functions provided by the user. We chose to instrument the functions because we measured that 91% (64 functions out of 70) of the functions automatically instrumented by RDAMcalculator are executed many times. For the remaining 6 functions, as they are executed few times, the probability of capturing an interference-free occurrence becomes low. Therefore, the RDAM score tends to lower the interference effect.

[1] We measure the time taken by 1000 executions of `rdtscp`. The large number of cycles comes from the flush of the pipeline.

For these potentially *false negative* functions, the developer can manually instrument an inner loop in order to capture more occurrences (manual instrumentation in Figure 4.2). We discuss in details the 6 manually instrumented functions in Section 5.2.3.

We also chose to only insert probes for a specific list of functions because instrumenting all the functions often leads to a large overhead. As illustrated in Figure 1, there are two ways to automatically generate this list. The developer can first generate this list with a where-oriented tool, such as `linux perf` or `Coz`. In this case, RDAMcalculator is used to try to identify all the functions that suffer interference before trying to understand why with cause-oriented tools. The developer can also generate this list with a cause-oriented tool. In this case, RDAMcalculator is used to verify that an interference bottleneck reported by the cause-oriented tool actually degrades performance and that spending time on trying to optimize the function is interesting.

4.2.2. False positives

RDAMcalculator reports functions as suffering interference when they have a variable execution time. In fact, we assume that two calls to the same function should always take roughly the same time, which is the time taken by the fastest, interference-free, occurrence. While this assumption holds for many functions, it may also lead to *false positives*. Some functions can take more time because of their workload: because of their 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.

Automatically identifying when the workload changes the execution time in general is difficult. First, static analysis does not help because static analysis cannot identify the 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. Then, even a varying workload can lead to approximately the same execution time. Typically, the time taken by a function that writes small buffers to a disk will take approximately the same time, even when the buffer size changes, because the function spends most of its time in the system call.

As handling a varying workload in general is difficult, we consider that reporting false positives remains the best strategy. In our experiment, we show that, among the functions with a high RDAM, RDAMcalculator only reports 20% of false positives. If RDAMcalculator reports a false positive, the developer can interactively specify which parameters matter before restarting the computation of the RDAM scores (feedback loop in Figure 4.2). Moreover, RDAMcalculator 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.

```
#timestamp  #ThreadId  #Event
0.00175s       1       Enter function Foo(arg1=17)
0.20573s       1       Enter function Bar(n=42.23)
0.21248s       2       Enter function Baz(a=21, b=40)
0.31054s       2       Leave function Baz(a=21, b=40) return value=91
0.61057s       1       Leave function Bar(n=42.23) return value=124.89
[...]
```

Figure 4.3: Example of a trace

4.2.3. Automatic instrumentation of the application

Since manually instrumenting an application can be a tedious task for the developer, RDAMcalculator automatically instruments the functions provided in the list of functions (step 1 of Figure 4.2). For that purpose, RDAMcalculator relies on the EZTrace tracing framework [58]. In detail, RDAMcalculator 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 intercepts the calls by patching the binary when the binary is loaded in memory by the ELF loader. In both cases, EZTrace replaces a call to the original function by a wrapper provided by RDAMcalculator. 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).

4.2.4. Trace generation

As presented in Figure 4.2 (2. Trace generation), RDAMcalculator runs the instrumented version of the application. The instrumented version records the events used to compute the RDAM 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. Figure 4.3 shows the textual display provided by our tool which allows a manual inspection of the detailed information of a trace. It is also possible to vizualise the trace graphically using a tool such as Vite [59] to get a first high-level view of the application's behavior. Figure 4.4 shows an example of the graphic view.

The instrumented version also records call stacks that lead to the execution of the instrumented functions because understanding the control flow of the application simplifies the analysis of the application.

As a large instrumentation overhead may lead to a drastically modified timing behavior and thus an inaccurate RDAM scores, we have carefully optimized the trace

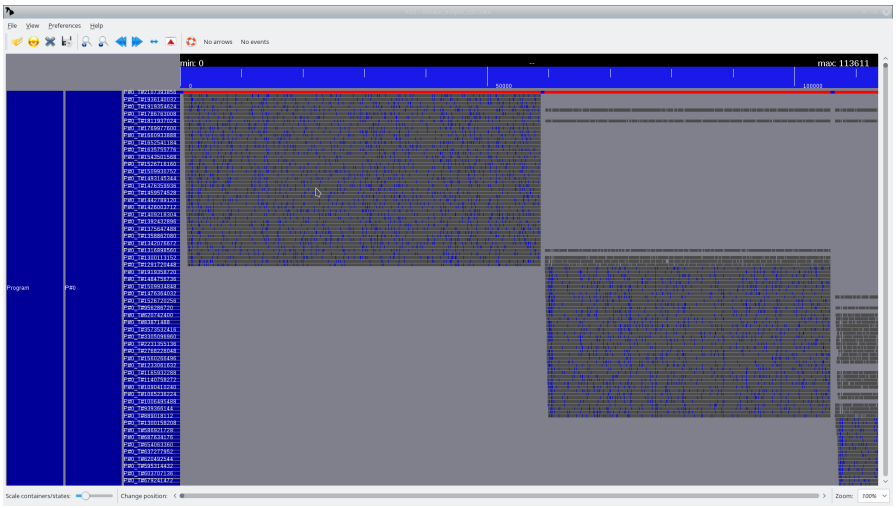


Figure 4.4: Trace vizualisation

generation mechanisms. We describe these mechanisms in detail in the remainder of this section.

Recording the events RDAMcalculator often records millions of events. For instance, the trace of the `DC` application contains 364 millions events. As a result, a trace is often large (17 GB for `DC`). Since recording this trace in a file can drastically change the timing behavior of the application, RDAMcalculator batches the I/Os. In detail, each thread of the application stores its events in its own pre-allocated buffer. A thread uses its own buffer so as to avoid thread synchronizations. When a buffer is full, RDAMcalculator flushes the buffer to disk and reinitializes the buffer. RDAMcalculator also flushes all 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 5.1), we pre-allocate 1 GB for the buffers (1/8 of the memory), while on our large machine (Opteron48, see Section 5.1), we pre-allocate 32 GB (also 1/8 of the memory).

Recording the call stacks In addition to the recorded events, RDAMcalculator 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, RDAMcalculator records a call stack every N invocations, by beginning with the first invocation. In our experiment, N is equal to 10 000 and is enough to understand why a function suffers interference.

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 5.1), we have measured that the overhead of an instrumented function always remains below 100 ns.

4.2.5. RDAM score computation

During the second phase, RDAMcalculator computes the RDAM scores. RDAMcalculator accepts the format used to store the execution trace during the in-vitro phase, but also the `Pajé` format, which makes it compatible with many other profiling tools [58, 60–62].

Before computing the RDAM scores, RDAMcalculator first identifies the repetitive sequences of events in the trace. RDAMcalculator 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:

a b c b d **a b** e **a b** c

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

As soon as RDAMcalculator identifies the repetitive sequences of instructions, it computes the RDAM score of each repetitive sequence with the formula given in Section 4.1. RDAMcalculator then reports the repetitive sequences of events. For each repetitive sequence of each thread, RDAMcalculator reports the name of the event (i.e., the function name), the RDAM score, the number of occurrences of the repetitive sequence and the recorded call stacks. The developer can then interactively change how RDAMcalculator handles the parameters of the functions and can thus discard the false positives before restarting the computation of the RDAM scores (feedback loop in Figure 4.2, see Section 4.2.2).

4.2.6. Conclusion

We use the RDAMcalculator to compute RDAM scores for repetitive sequences of instructions in an application. The obtained RDAM scores indicate whether a given sequence suffers interference. The next chapter presents the two-step experimental study through which we validate our approach.

5. EVALUATION

This chapter aims to evaluate the RDAM metric through a two-step experimental study. First, we use several micro-benchmarks to study the correlation between an RDAM score and the performance of the sequence of instructions associated with it. Second, we study 27 real applications to evaluate the effectiveness of RDAM in detecting previously known as well as new performance problems in them.

5.1. 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 metric actually captures interference effects. For that purpose, we evaluate well-known problems: lock contention (POSIX and spinlock), 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 the RDAM score is correlated to the performance of the micro-benchmark.

Second, as presented in Section 4.1, we suppose that the probability of recording an execution not slowed down by interference is high when we record many occurrences of a repetitive sequence of instructions. 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.

5.1.1. 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

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

Listing 5.1: Code of the lock contention micro-benchmark

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

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

Listing 5.2: 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 5.3: Code of the I/O contention micro-benchmark

lock at the same time. In case of contention, the cache line that holds the lock variable continuously bounces between the cores, which may saturate the buses between the cores (e.g., the interconnect on Opteron48). As presented by Lozi et al. [40]., this saturation drastically increases the time to acquire a lock and leads to a performance collapse.

Listing 5.1 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 leave an idle core so that the OS can schedule other ready processes), and we vary `delay` to simulate different levels of lock contention.[1]

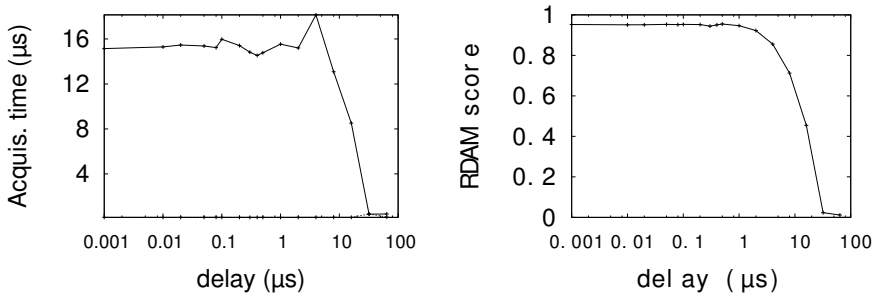
The third micro-benchmark suffers false sharing. False sharing appears when multiple threads access different variables that happen to be located on the same cache line [29,44]. Each thread, by updating its own variable, invalidates the cache line for the other threads, which leads to cache misses and 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 5.2 reports the code used to evaluate false sharing. The first thread (`my_rank = 0`) continuously updates its variable `x`. The other thread (`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.

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.

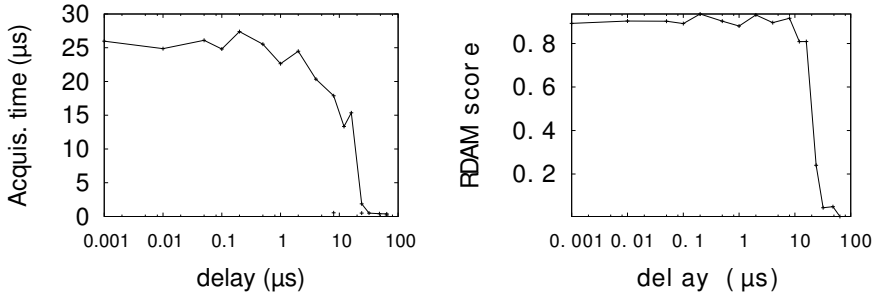
Listing 5.3 reports the code used to evaluate I/O contention. Each thread opens its own file and reads it sequentially with a delay between read operations. In order to bypass the I/O cache of the operating system, each thread opens its file with the `O_DIRECT` flag, which ensures that every call to `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.

[1] Note that in these micro-benchmarks, as in all the other micro-benchmarks, the delay is not included in the profiled sequence of instructions.



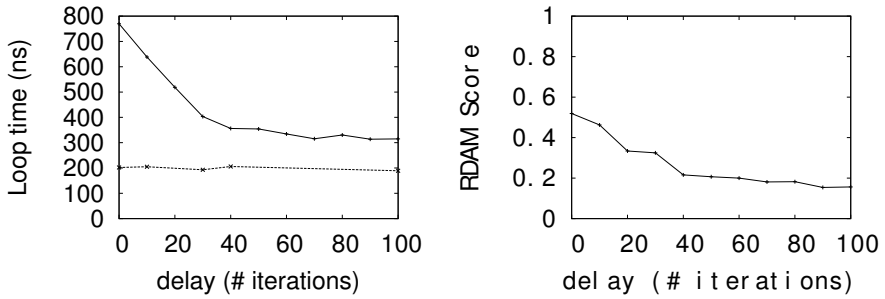
(a) Acquisition time (b) RDAM score

Figure 5.1: Lock contention micro-benchmark (POSIX lock)



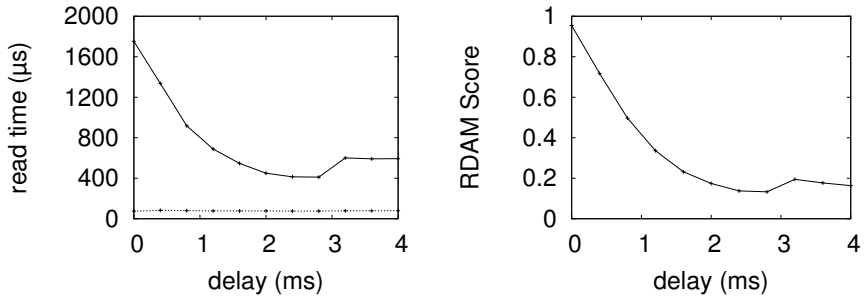
(a) Acquisition time (b) RDAM score

Figure 5.2: Lock contention micro-benchmark (spinlock)



(a) Read time (b) RDAM score

Figure 5.3: False-sharing micro-benchmark



(a) Read time (b) RDAM score

Figure 5.4: I/O contention micro-benchmark

Benchmark	X	ρ	# samples
POSIX lock contention	lock acq. time	0.97	17
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 5.1: Correlation coefficient for the micro-benchmarks.

5.1.2. Analysis of the micro-benchmarks

Figures 5.1, 5.2, 5.3 and 5.4 report the evaluation of the micro-benchmarks. In each figure, (a) gives the performance (solid line) and the minimal duration (dotted line) when we vary delay, and (b) 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 5.1 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 coefficient 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 occurrence only slightly slowed down by interference in each of the micro-benchmarks, even when the contention level is extremely high. This result validates the hypothesis formulated in Section 4.1: when the number of occurrences of a repetitive sequence is large, the probability of capturing an occurrence that suffers little interference is large.

	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	3	2	1	3	1	1	11
# functions with a high score	3	3	1	6	2*1/2	1	15
# false positives	2	0	0	1	0	0	3
# applications with true defects	1	2	1	3	1	1	9
# functions with true defects	1	3	1	5	1	1	12
# functions never reported	0	1	0	4	0	1	6
# corrected functions	1	3	0	5	1	0	8

Table 5.2: Summary of the evaluated applications

5.2. APPLICATIONS EVALUATION

This section presents an evaluation of RDAMcalculator with real applications. This evaluation has first the goal of verifying that RDAMcalculator can actually identify the effects of interference in real applications. For that purpose, we use RDAMcalculator to evaluate a panel of 27 applications. This evaluation also aims at showing that we can cross-check the interference bottlenecks identified by RDAMcalculator with the reports of classical cause-oriented tools in order to fully understand interference (where, why and how much). This evaluation has finally the goal of showing that, by identifying where, why and how much interference impacts performance, 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 RDAMcalculator. 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 RDAM score and an analysis of the false positives reported by RDAMcalculator.

5.2.1. Evaluated applications

We evaluate 27 applications summarized in Table 5.2. We have selected these applications because they are widely used in the validation process of tools and metrics designed for performance analysis of parallel programs. Moreover, some of them are already known to suffer interference due to various contention types such as lock contention and false sharing [40, 41, 44, 63, 64].

Phoenix-2 [21] is 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 [20] contains small multi-threaded C applications, ranging from a ray tracer to a large-scale ocean movement computation. The Parsec 2.1 benchmark [22] contains moderate to large multi-threaded C++ applications from various fields such as financial analysis or data-mining. NPB (NAS Parallel Benchmark 3.3 [23]) contains moderate

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

Table 5.3: Time consuming functions reported by `Linux perf`

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 [65]. We use the large C class dataset. While the other benchmarks synchronize with POSIX locks and condition variables, NPB synchronizes through OpenMP.

Memcached 1.4.36 [24] is an in-memory key-value caching system widely used to speed up web servers by alleviating database load. We evaluate memcached with the memaslap client [66], which generates 70% of set requests and 30% of get requests 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 [25] 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.

5.2.2. Identification of the hottest functions

As presented in Section 4.2.1, RDAMcalculator automatically computes the RDAM scores of a list of functions provided by the developer. This list can contain a list of functions identified by a cause-oriented tool in order to verify that a reported interference has actually an important performance effect. The list can also contain a list of functions identified as potentially interesting by a where-oriented tool in order to identify all the functions for which interference has a large performance effect.

In our experiments, we analyze the functions reported by a where-oriented tool because we want to identify all the possible interference effects. 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. For that purpose, we use the where-oriented tool `linux perf` that computes the time spent in the functions with a sampling technique. Table 5.3 reports, for each application, the functions that take more than 1% of the total execution time. In total, we thus analyze 70 functions from 27 applications with RDAMcalculator.

5.2.3. Analysis of the potential false negatives

As presented in Section 4.2.1, if a function is called few times, the probability of capturing an interference-free execution becomes low. In this case, the RDAM score is low because RDAMcalculator 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 5.3). These functions are called once. The RDAM score is thus systematically equal to 0, which may hide an interference bottleneck.

For these 6 potential 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

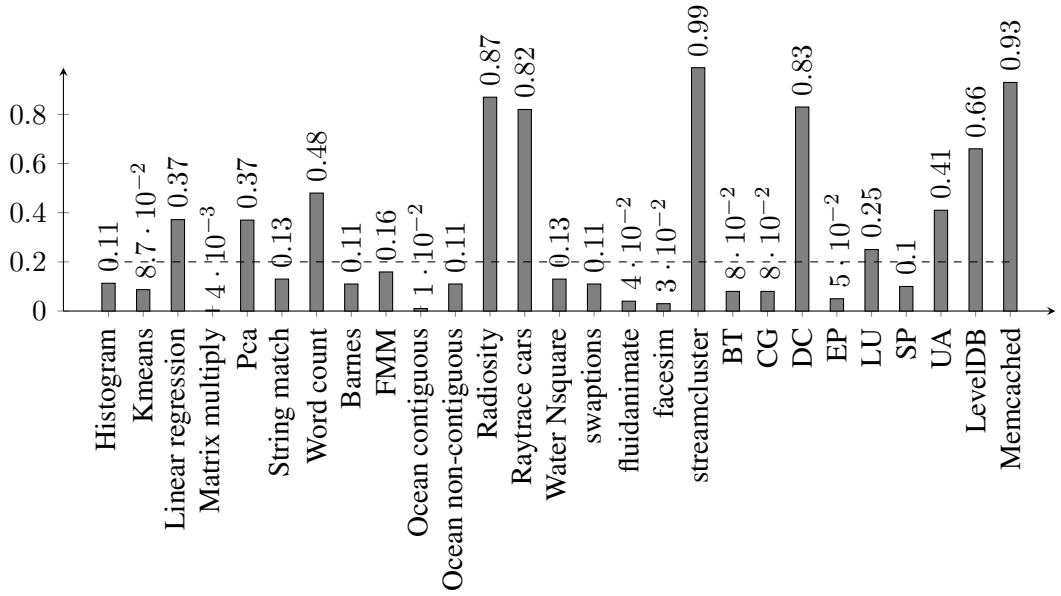


Figure 5.5: Highest RDAM scores

to a larger slowdown. For each function, we manually add 2 lines of code: one to count the number of iterations and the other to record the timestamp.

5.2.4. Instrumentation overhead

The last column of Table 5.3 reports the overhead caused by instrumentation (manual instrumentation for the 6 potential false negatives and automatic instrumentation otherwise). The overhead remains below 10% for 22 applications, and below 30% for the other applications. We consider that this overhead remains reasonable for an in-vitro profiling, and that it should not drastically change the behavior of the applications.

5.2.5. Analysis of the RDAM scores

Figure 5.5 reports the highest RDAM scores identified by RDAMcalculator. For the analysis, we focus on applications with an RDAM score greater than 0.2. With this threshold, a thread loses more than 20% of its time because of interference. We consider thus that it becomes interesting to understand why. We identify 15 functions from 9 applications with a RDAM score higher than 20%. The remainder of the section presents an exhaustive analysis of these functions.

Lock contention RDAMcalculator reports high RDAM scores for the lock acquisition function in two applications: Raytrace (RDAM of 0.82) and Radiosity (RDAM of 0.87).

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

Listing 5.4: code of the hottest function in Raytrace

Thanks to the call stacks reported by RDAMcalculator, we can easily identify where is the interference bottleneck in the code. Listing 5.4 presents the code associated with the lock acquisition in Raytrace.

As presented in Section 5.1.1, when many threads try to acquire the lock at the same time, the cache line that contains the lock bounces between the cores, which saturates the buses. A high RDAM score in the lock acquisition function is the symptom of this saturation. Lozi et al. [40,41] 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 RDAM 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.

False sharing RDAMcalculator identifies two functions that suffer false sharing, one from Raytrace that was never reported (RDAM of 0.25), the other from Linear_regression that was previously reported by Liu et al. [44] (RDAM of 0.37).

The high RDAM score in Raytrace appears after correcting the lock contention (see Section 5.2.5), and in the same function (see Listing 5.4). 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 5.4. However, the RCL lock should prevent these cache misses, because only the RCL server core accesses `gm->rid` [40]. 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 5.2.5), 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 RDAM scores in both applications.

Network configuration	RDAM score		Transaction/s	Network rate
	client	server		
100 Mb	0.93	< 0.1	36k	11 M/s
1 GB	0.82	< 0.1	193k	61 M/s
Local loop	0.62	< 0.1	2 037k	632 M/s

Table 5.4: Correlation between the network saturation and the RDAM score in memcached.

Network contention When RDAMcalculator automatically instruments the time consuming functions of memcached, RDAMcalculator reports very low RDAM scores. This result shows that, in our experiment, memcached does not seem to suffer 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 5.4 presents the result for different network configurations: a network at 100 Mb, a network at 1 Gb and a local network loop when we co-localize the client and the server on a 32-core Intel Xeon E5-1607. The low RDAM score of the server with each configuration confirms that the server does not suffer interference. Moreover, we can see that the RDAM scores at the client side decrease when the network contention decreases. These results clearly show that the network suffers interference, and that the RDAM score accurately identifies the network contention. We can also see that the RDAM score remains high when we co-localize both the server and the client on the same machine, which shows that the network remains a bottleneck even in this case.

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 found that the application inserts new keys in mutual exclusion by using this variable condition function. 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 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 RDAM score of 0.99 (already reported in [63, 64]). 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 RDAM score is high because in some rare cases a thread crosses the barrier quickly. Correcting this interference 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 interference bottleneck would require a large code rewriting.

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.

While LU was studied in other works that target NUMA architectures [31,67], it was not identified as problematic. We suppose that our software setting is slightly different (Linux, gcc or glibc versions), which explains why a NUMA bottleneck appears in our experiment.

For LU, RDAMcalculator generates a trace with 30.8 million events (1.3 GB). RDAMcalculator reports 3 functions with a high RDAM score: `sync_left` (RDAM of 0.25), `rhs` (RDAM of 0.24), and `buts` (RDAM of 0.20).

In order to understand why these functions suffer 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 pined 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 RDAM 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 all the high RDAM scores and the memory imbalance. The completion time thus drops from 101s to 64s.

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, RDAMcalculator generates a trace with 364.7 million events (17 GB). RDAMcalculator reports two functions with a high RDAM score: `MultiWayMerge` (RDAM of 0.83) and `_write_nocancel` (0.33). `MultiWayMerge` is a false positive discussed in Section 5.2.6.

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, we have verified that the I/O stack suffers interference 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.

5.2.6. Analysis of the false positives

Overall, we found 3 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.

Word_count In word_count, RDAMcalculator 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 5.5 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.

```
while (i < num_rows) {
  for (j = i; j < num_rows; j++)
    compute_cov_matrix(i,j);
  i = next_row();
}
```

Listing 5.5: Highly varying loop in pca

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.

5.2.7. Conclusion

Our experiments show that the RDAM score accurately assesses how much interference impacts performance, regardless of the causes (lock, parallelism, false sharing, NUMA placement, disk and network). Our study also shows that, by cross-checking with cause-oriented tools, we can explain where, why and how much interference impacts performance. Moreover, our experiment shows that we can remove some of the interference bottlenecks by modifying few lines of code. Finally, we found 3 false positives and we show that they were relatively easy to identify.

6. Conclusion

Whether directly or indirectly, applications' performance affects our daily life. It is as important to digital service providers as it is to the consumers of those services. With the complexity of modern architectures, getting satisfactory performance is getting harder. Actually, even performance analysis is rather complicated for parallel applications. 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. The average developer is not capable of producing quality multi-thread code which is both correct and efficient. To help developers locate sources of inefficiency in their code, new performance analysis techniques are needed.

There is already a multitude of performance analysis tools, most of which are cause-oriented. In other words, these tools are specifically designed to detect a problem in particular. Such tools are very effective at finding exactly one kind of interference. A few other tools opt for genericity and have their own perception of what a performance problem (optimization opportunity) mostly looks like: for instance a function globally taking a long time, or whose hypothetical optimization is guaranteed to reduce the program's completion time. These symptoms do nonetheless not necessarily indicate the presence of a real problem. In sum, the perfect tool that automatically diagnoses and fixes a parallel program does not exist. Even trying to combine the results from existing tools does not provide a systematic approach to tackle the problem of performance analysis.

This thesis proves the relevance of a point missed by most previous tools, which is quantifying performance impact on an application due to interference. Measuring performance cost regardless of the cause can be the focal point in a performance investigation since it directly indicates the presence of a real problem on the one hand and can guide the specialized analysis (by a cause-oriented tool) on the other hand. Moreover, this thesis shows that performance variation is a very accurate indicator of performance loss due to interference.

Based on these observations, we propose to decouple the analysis of the interference causes from the analysis of the interference effects. We propose RDAMcalculator to identify the interference effects, and we show that it highlights interference, regardless of the interference causes. Our experiment with micro benchmarks and applications shows that RDAMcalculator successfully detects interference with few and easy to dis-

card false positives. Our experiments also show that, by cross-checking the reports of RDAMcalculator with the reports of cause-oriented tools, we can fully understand interference: we can identify the blocks of code that suffers interference (where), we can explain why each block of code suffers interference, and we can measure how much interference degrades the performance of each block of code.

These findings bring new answers to the problem of performance analysis, but also bring to mind new questions and perspectives.

6.1. FUTURE WORK

Our work could be extended in several ways. For instance, testing our approach on Java parallel applications can be interesting knowing that such applications are widely deployed in data centers and are in need of scalability. Our current instrumentation technique is not fully adapted to handle such applications. A study of the available tracing mechanisms for Java byte code would be helpful to get a better understanding of how they could be used in combination with RDAMcalculator or potentially implement our own tracing facility in the Java virtual machine.

Another track of interest can be the automatic detection of false positives before reporting the results to the user. For example, functions showing a high RDAM score in a first reporting round can be instrumented a second time while recording, in the scope of these functions, hardware performance counters that can help demonstrate a variable number of instructions executed by the function across the different calls. However, in a number of cases, a varying number of retired instructions does not indicate a false-positive. This is the case of busy-waiting algorithms, for which the number of retired instructions increases with the contention (e.g., pthread_spinlock, MPI_Wait, lock free algorithms, etc.). A possible solution to reduce the number of this kind of false negatives is to integrate, directly in the tool, a list of well-known functions with such behavior. Nonetheless, we still have to deal with the case where a "normal" function calls functions of this type: how should we interpret a varying number of retired instructions in this situation? We may think of designing a few micro-benchmarks to try to characterize the variation of the number of instructions in the case of a busy waiting in comparison to variation due to parameters.

Before investing more time investigating this aspect, one can ask the following question: is the false positives issue really a burden for the user? In our experiments, we have instrumented 70 functions and found 3 false positives. This result seems statistically representative. For this reason, we suspect that profiling a bigger system such as the JVM should not lead to a larger false positive rate (we will have easy to identify cases such as the object copy in the GC for example). However, one may want to check this for sure by doing a study of false positives in a production system. If conclusive, such study would tell us whether we really should invest more effort in automatically

detecting functions whose performance is parameter-dependent.

Furthermore, to be able to leverage our approach with large distributed client-server applications in practice, it can be interesting to be able to automatically instrument event-based frameworks so that RDAMcalculator can automatically generate RDAM scores for asynchronous requests.

For the time being, our instrumentation mechanism incurs a lower than 11% overhead in most cases. In a few cases, the overhead reaches 30% which can be considered fairly high. As a consequence, two questions arise: 1) does a high overhead change the application's behavior such that our tool may fail to detect some performance bugs? 2) This kind of overhead becomes bothersome in large production systems. So, how can we mitigate it? For the first question, one way would be to measure, for each program, both the original and instrumented program many times and see how the variances compare. If both programs show the same sort of variance, it suggests that the instrumentation is preserving the program behavior. For the second question, a possible solution is to resort to sampling instead of measuring every occurrence of the instrumented functions. Of course, sampling means a loss of information which leads to some rate of inaccuracy. However, there are some research results that can be used so as to minimize the potential bias incurred by sampling.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Pradeep Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@ Intel Magazine*, 9(2):1–10, 2005.

[2] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael 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*, pages 603–618, 2016.

[3] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the conference on Networked Systems Design and Implementation, NSDI'12*, pages 26–26, 2012.

[4] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'10*, pages 269–280, 2010.

[5] Xiao Yu, Shi Han, Dongmei Zhang, and Tao 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*, pages 193–206, 2014.

[6] Florian David, Gaël Thomas, Julia Lawall, and Gilles 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*, pages 291–307, 2014.

[7] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'10*, pages 739–753, 2010.

[8] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce 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*, pages 28–36, 2008.

[9] Stephan M Günther and Josef Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33, 2009.

[10] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the international conference on Virtual Execution Environments, VEE'11*, pages 27–38, 2011.

[11] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the International Symposium on Computer Architecture, ISCA'13*, pages 511–522, 2013.

[12] Marco Hobbel, Thomas Rauber, and Carsten 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.

[13] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. PREDATOR: Predictive false sharing detection. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'14*, pages 3–14, 2014.

[14] Tongping Liu and Xu Liu. Cheetah: detecting false sharing efficiently and effectively. In *Proceedings of the international symposium on Code Generation and Optimization, CGO'16*, pages 1–11, 2016.

[15] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine 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*, pages 1–12, 2013.

[16] Marc Casas and Greg 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*, pages 165–174, 2014.

[17] Perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[18] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'15*, pages 184–197, 2015.

[19] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, pages 561–578, 2014.

[20] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture, ISCA'95*, pages 24–36, 1995.

[21] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the symposium on High Performance Computer Architecture, HPCA'07*, pages 13–24, 2007.

[22] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'06*, pages 72–81, 2008.

[23] Nas parallel benchmark 3.3. https://www.nas.nasa.gov/Software/NPB/.

[24] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[25] Sanjay Ghemawat and Jeff Dean. LevelDB. *URL: http://leveldb.org*, 2011.

[26] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the International Conference on Parallel, Distributed, and Network-Based Processing, PDP'10*, pages 180–186, 2010.

[27] Christiane Pousa Ribeiro, Jean-Francois Mehaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 59–66. IEEE, 2009.

[28] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM, 2010.

[29] ML Scott and WJ Bolosky. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, page 57, 1993.

[30] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling dynamic openmp applications over multicore architectures. *OpenMP in a New Era of Parallelism*, pages 170–180, 2008.

[31] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark 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*, pages 381–394, 2013.

[32] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 9. IEEE Press, 2008.

[33] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony IT Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *OSDI*, volume 8, pages 15–28, 2008.

[34] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[35] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[36] Jerzy Cetnar, W Gudowski, and J Wallenius. Mcb: A continuous energy monte carlo burnup simulation code. *Actinide and fission product partitioning and transmutation*, 1999.

[37] Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[38] Claude Bernard, Tom Burch, Thomas A DeGrand, Carleton DeTar, Steven Gottlieb, Urs M Heller, James E Hetrick, Kostas Orginos, Bob Sugar, and Doug Toussaint. Scaling tests of the improved kogut-susskind quark action. *Physical Review D*, 61(11):111502, 2000.

[39] Yang Wang, GM Stocks, WA Shelton, DMC Nicholson, Z Szotek, and WM Temmerman. Order-n multiple scattering approach to electronic structure calculations. *Physical review letters*, 75(15):2867, 1995.

[40] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles 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*, pages 65–76, 2012.

[41] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Fast and portable locking for multicore architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(4):13:1–13:62, 2016.

[42] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[43] Jie Tao and Wolfgang Karl. Cachein: a toolset for comprehensive cache inspection. In *Proceedings of the International Conference on Computational Science, ICCS'05*, pages 174–181. 2005.

[44] Tongping Liu and Emery 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*, pages 3–18, 2011.

[45] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew 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*, pages 141–154, 2013.

[46] Jack Dongarra, Kevin London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You, and Min 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*, pages 289.2–, 2003.

[47] Vince Weaver. The unofficial linux perf events web-page, 2013.

[48] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2013.

[49] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'10*, pages 335–348, 2010.

[50] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the international symposium on Code Generation and Optimization, CGO'10*, pages 257–265, 2010.

[51] Chi Xu, Xi Chen, Robert Dick, and Zhuoqing Morley 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*, pages 76–86, 2010.

[52] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'16*, pages 3:1–3:14, 2016.

[53] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM, 2011.

[54] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[55] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*, 2012.

[56] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'06*, pages 89–102, 2006.

[57] Stijn Eyerman, Kristof Du Bois, and Lieven 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, pages 145–155, 2012.

[58] François Trahay, Yutaka Ishikawa, François Rue, Raymond Namyst, Mathieu Faverge, and Jack Dongarra. Eztrace: a generic framework for performance analysis. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing, CCGRID'11*, pages 618–619, 2011.

[59] K Coulomb, M Faverge, J Jazeix, O Lagrasse, J Marcoueille, P Noisette, A Redondy, and C Vuchener. Visual trace explorer (vite). Technical report, Technical report, 2009.

[60] Lucas M. Schnorr. Poti. https://github.com/schnorr/poti.

[61] Lucas M. Schnorr. Akypuera. https://github.com/schnorr/akypuera.

[62] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Proceedings of the international conference on Computer Modeling and Simulation*, pages 126–131, 2008.

[63] Gabriel Southern and Jose Renau. Analysis of PARSEC workload scalability. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'16*, pages 133–142, 2016.

[64] Mark Roth, Micah J Best, Craig Mustard, and Alexandra Fedorova. Deconstructing the overhead in parallel applications. In *Proceedings of the International Symposium on Workload Characterization, IISWC'12*, pages 59–68. IEEE, 2012.

[65] Michael A Frumkin and Leonid V Shabanov. Benchmarking memory performance with the data cube operator. Technical report, NASA, 2004.

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

[67] Gauthier Voron, Gaël Thomas, Vivien Quéma, and Pierre Sens. An interface to implement NUMA policies in the xen hypervisor. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'17*, page 14, 2017.