Internship Report

Improving the Performance of Java Applications on Multicore Architectures

By :

Jean-Pierre Lozi
jean-pierre@lozi.org

Under the direction of :

Gaël Thomas	Gilles Muller	Julia Lawall
gael.thomas@lip6.fr	gilles.muller@lip6.fr	julia@diku.dk

September 8, 2010

Contents

1. Introduction 2

2. Related work 2

2.1. Cache locality . . . 2

2.2. Contention management . . . 3

2.2.1. Limiting the sharing of global data structures . . . 3

2.2.2. Optimizing locks . . . 4

2.3. Efficient inter-process communication . . . 4

2.3.1. Shared memory . . . 5

2.3.2. Message-based communication . . . 5

2.4. Fast transfer of control . . . 7

2.5. Heterogeneous architectures . . . 9

2.6. Conclusion . . . 10

3. Experiments 10

3.1. Rough estimates . . . 11

3.1.1. Cost of cache misses . . . 11

3.1.2. Cost of transfers of control . . . 12

3.1.3. Cost comparison . . . 13

3.2. NULL RPCs with multiple clients . . . 13

3.3. Final benchmark . . . 17

3.3.1. Executing critical sections locally . . . 17

3.3.2. Context and shared variables . . . 18

3.4. Future experiments . . . 19

4. Conclusion 19

Bibliography 20

Appendix: source code 21

1. Introduction

Multicore architectures are becoming increasingly common in domains ranging from personal computing to server applications and even embedded systems. Exploiting the power of additional cores is a challenging issue, for several reasons. First, writing parallel programs is harder than writing sequential programs, and even state-of-the-art parallel programs comprise sequential sections that constitute a major limiting factor to overall performance. Second, synchronization between threads can cause major overhead. Using fine-grained synchronization primitives can limit this overhead, albeit at the cost of a greater complexity, lower maintainability and lower extendability of the source code. Finally, low-level cache coherence algorithms are costly, especially when the locality of the shared data between threads is low.

Exploiting the computing power offered by a large number of processing cores is especially hard for programs that are run within Java Virtual Machines (JVMs). Indeed, in Java, the only concurrency management mechanism is the synchronized block, which does not allow for low-level optimizations (e.g. improving data locality between threads or limiting contention via fine-grained locking). Such optimizations have to be implemented within the virtual machine itself.

In the long term, we plan to experiment with a novel approach to improve the performance of legacy Java applications on multicore hardware. The general idea is to improve data locality between cores in order to limit data transfers caused by cache coherency algorithms. Since Java synchronized blocks are used to ensure coherency when accessing shared variables, they can be used as delimiters of code areas that use shared data. Our idea, based on this observation, is to dedicate a number of processing cores to the execution of synchronized blocks in order to improve performance: indeed, these dedicated cores will be likely to have most of the shared data readily available in their local cache, most of the time. Since the majority of architectures will feature a large number of cores in the near future, dedicating a few cores to a dedicated task should not have a major negative impact on performance.

There is however a tradeoff between the increased locality of shared data and the loss of locality caused by the need to fetch context variables (i.e. thread-local variables used in synchronized blocks) from the original processing core. Furthermore, executing a synchronized block on a dedicated core requires a transfer of control to this core. This is not the case in the traditional way of executing Java programs, in which all synchronized blocks are executed locally. We therefore have to figure out when the cost of the transfer of control is covered by the performance gain caused by the availability of data in the cache memory of the dedicated core.

In the context of this internship, we focus on evaluating the tradeoff between executing synchronized blocks locally and migrating them to improve data locality. Preliminary results show that our technique is profitable on a 8-core microprocessor with a moderate number of shared cache lines accessed by each critical section (>15 cache lines), if a reasonable number of context variables are used. These encouraging results lead us to decide to further pursue this analysis in the context of a PhD thesis, in which we plan to implement the automatic migration of critical sections towards dedicated cores in a Java Virtual Machine.

The remainder of this report is organized as follows. In Section 2, we present chosen works from the research literature related to the main issues we are faced with in the context of our project. In Section 3, we present the experiments we performed in the context of this internship. We conclude in Section 4.

2. Related work

In this section, we analyze the existing work which, to our knowledge, best tackles the major issues raised by our project. Since, in the context of our project, we aim to evaluate the tradeoff between the performance gain caused by improved data locality and the overhead caused by transfers of control on multicore architectures, the major issues we will have to deal with will be performance-related. This is especially true since multicore architectures raise complex issues regarding performance: programmers who write code for these architectures are often faced with problems such as bottlenecks caused by contention, poor locality of shared variables, and costly transfers of control and data between cores.

In Section 2.1, we focus on cache locality issues. Then, in Section 2.2, we focus on the major cause of overhead in multicore architectures : contention. In Section 2.3, we raise the issue of inter-process communication. Section 2.4 is dedicated to fast remote execution of code between cores. In Section 2.5, we discuss ways to improve performance on heterogeneous architectures. Finally, we conclude this analysis in section 2.6.

2.1. Cache locality

In this section, we focus on cache locality issues, about which two recent works have provided us with considerable insight: Pesterev et al.'s DProf [13] and Koufaty et al.'s bias scheduling [11].

DProf

DProf is a profiling tool that dynamically tracks indicators associated with data structures instead of code locations in order to allow for more efficient localization of cache-related bottlenecks. It offers four views of the collected data: the *Data Profile view* shows the number of cache misses for each of the most common data types; the *Miss Clarification view* shows which types of misses (due to sharing, associativity or capacity overload) are most common for each data type; the *Working Set view* indicates which data types were most active, how many of each were active at any given time and the cache associativity sets used by each data type; finally, the *Data Flow view* shows the most common sequences of functions that reference particular objects of a given type.

To create these views, DProf collects two kinds of data: *path traces* and *address sets*. A path trace records all accesses to a single data object. An address set is a data structure that contains the address and type of every object allocated during execution. Both of these data structures are built from the two kinds of raw data ultimately collected by DProf: *access samples* and *object access histories*. An access sample records information for randomly

chosen memory-referencing instructions, and an object access history is a complete trace of all instructions that read or write a particular data object, from when it was allocated to when it was freed. To collect access samples, DProf uses AMD's proprietary Instruction Based Sampling (IBS), but similar facilities are available on Intel chips.

Pesterev et al. show that, in two case studies, DProf provides much more helpful information than both lock_stat (a profiling tool that reports statistics about locks) and OProfile (a traditional execution profiler). Furthermore, performance experiments show that DProf's overhead varies depending on several variables (IBS sampling rate, number of debug register interrupts triggered by second, etc.). However, it is usually reasonably low and remains lower than 15%.

DProf is interesting to us at two levels. On a first level, the tool itself will probably be useful to monitor bottlenecks if we do not manage to compensate the cost of transfers of controls by the performance increase caused by cache locality efficiently enough in our project. On a second level, DProf's implementation itself is extremely interesting, since it uses special CPU instructions to monitor cache misses in order to infer cache locality penalties. Pesterev et al. provide advanced information on how they use these instructions in their article; we will not, however, discuss this point into more details in this document, for fear of getting too technical. The reader is invited to directly refer to Pesterev et al.'s paper [13] for more information.

Bias scheduling

Koufaty et al. [11] propose to optimize the scheduling of processes on performance-asymmetric multicore architectures with an algorithm that evaluates the performance gain of running an application on a fast core rather than on a slow core. Their approach estimates the number of internal stalls, caused by local (internal to the core) cache misses and delays, and external stalls, caused by accesses to shared last level caches, memory and I/O. To estimate internal stalls, they compute the number of cycles during which the front-end of the machine is not delivering instructions to the back end, thus effectively measuring the time during which cores are idling due to a lack of instructions. To estimate external stalls, they measure the number of requests serviced outside the core. The details of these mechanisms are not clearly stated in their paper, but we plan to analyse their implementation. We hope that this will allow us to find out how to obtain efficient statistics about cache misses that we will use to compute our cache locality penalties.

Now that we have considered cache locality issues directly, we will focus on the related problem of contention management. Indeed, cache locality issues are often linked with contention, since a lot of cache misses are caused by access conflicts over data stored in a given cache line. This is the object of the next subsection.

2.2. Contention management

Contention is a major issue on the target architecture for our project, i.e machines that feature a large number of cores. Migrating critical sections to remote cores will require the use of shared structures for synchronizations and data transfers; therefore, we must find ways to limit contention since, in the context of our project, our main objective is to ensure good performance.

In this subsection, we first discuss strategies to limit the size of critical sections to limit the use of shared data structures. Finally, we focus on the locks themselves, whose efficiency is crucial to any multithreaded environment.

2.2.1. Limiting the sharing of global data structures

The most obvious way to limit contention is to limit the sharing of global data structures. Wickizer et al. wrote an operating system named Corey [15] that aims to be very efficient on architectures having a large number of cores. Corey limits the sharing of kernel data structures that are shared among cores in order to improve performance.

More precisely, Corey's motto is that *applications should control sharing*. Indeed, traditional operating systems tend to share a lot of data structures between kernel and user threads by default, which can lead to contention. Wickizier et al. argue that the kernel should arrange each data structure so that only a single processor needs update it, unless directed otherwise by the application.

Corey provides three basic operating system abstractions to allow applications to control inter-core sharing and to take advantage of architectures having many processing cores by dedicating some of them to specific operating system functions. These abstractions are *address ranges*, *kernel cores* and *shares*.

Address ranges allow applications to selectively share parts of address spaces, instead of being forced to make an all-or-nothing decision. Indeed, traditional OSes give only two choices for shared memory: applications can either use a single address space shared by all threads, or a separate address space per thread. This can be inefficient if only a subset of the threads need to access a data set. Address ranges are more flexible and thus allow for finer-grained optimizations.

Kernel cores allow applications to dedicate cores to kernel functions or data. For instance, a core can be devoted to interacting with a given device. Multiple cores can then communicate with this dedicated core via shared-memory IPCs when they need to interact with the device. In a traditional OS, each core would directly interact with the device, using locks to access these data structures concurrently, which would drastically impede performance due to contention. With Corey's architecture, there is no direct contention over the data structures of the device driver. This limits the overhead caused by locks.

Shares allow applications to dynamically create lookup tables for system data structures and determine how these tables are shared. Indeed, many kernel operations involve looking up identifiers in tables to obtain pointers to internal kernel data structures. The use of these tables can be costly due to contention: in mainstream OSes, they are usually either shared between all the threads of a process (this is typically the case for Unix file descriptors, for instance) or between all processes (this is typically the case for Unix process identifiers). With Corey, these tables are shared only between the cores that need them: each of an application's cores starts with one share (its *root share*), which is private to that core. Then, if two cores wish

to share a share, they create one and add its ID to their private root share (or to a share reachable from their root share). A root share doesn't use a lock (since it is private), but a shared share does. When an application requests the creation of a new kernel object, it decides which share will hold the identifier.

Thus, a share maps application-visible identifiers to kernel data structures. Each core can use all the shares reachable from its root share. Contention only arises when two cores manipulate the same share, and overhead due to contention can be greatly limited by constricting the sharing scope of shares that are only reachable by a subset of the cores.

Performance experiments show that Corey performs 25% faster than Linux when using 16 cores on a MapReduce task and a Web Server task. Hardware event counters show that these improvements are due to Corey's original handling of shared memory.

Corey's idea of arranging data structures so that a single processor accesses it is reminiscent of our project, we plan to migrate synchronized blocks to dedicated cores in order to limit the number of processors that access shared data (ideally, one, if there is a single dedicated core). However, in our project, we will be able to detect where shared variables are accessed (thanks to synchronized blocks); this is not the case with Corey in which client applications can access shared variables anywhere.

Other projects have focused on the concept of limiting shared data to improve performance. Baumann et al. [2] propose a new type of operating system named the *multikernel* which uses one kernel per core with all of the kernel data structures replicated on each core. This is another way to limit the sharing of global data structures. Actually, multikernels limit the sharing of data structures so much that they completely prohibit shared memory. Fähndrich et al. [6] propose another OS that follows this same paradigm. Not sharing data structures at all goes much further than limiting the sharing of data structures: this is a matter of inter-process communication. We will discuss these issues more in detail in Section 2.3. Before discussing these matters that bring us well beyond traditional contention management mechanisms, we will focus on ways to optimize the most common tool in traditional contention management techniques : the lock.

2.2.2. Optimizing locks

Another way to optimize multithreaded applications is to optimize the internal implementation of locks. Two opposing locking strategies are generally used: to wait for a lock to be released, a thread can either spin, actively polling a resource in a loop; or block, i.e. putting the thread to sleep and trusting the scheduler to wake it up when needed.

Spinning provides high reactivity (no context switch needed, instant detection of lock releases) but wastes significant CPU resources. Moreover, on an overloaded system using spinlocks, the scheduler often preempts lock holders to wake waiting threads up; these threads then waste their time slices actively polling a resource that cannot be released. This phenomenon can drastically impede performance.

Block-based approaches usually fare better that spinlocks under high load but their lack of reactivity introduces high overheads on the critical path of computation, thereby increasing the likelihood that other threads will encounter contention and block. This causes a vicious cycle of extremely slow lock handoffs known as a *convoy*. This is especially programs that use fine-grained locking since their critical sections usually take less time to execute than a single context switch.

Hybrid solutions that somewhat improve performance have been developed, but they still perform rather poorly, simply offering other tradeoffs between reactivity and CPU usage. However, Johnson et al. [10] proposed an interesting alternative to spinlocks and blocking locks that does not seem to suffer from this issue.

According to Johnson et al., contention management and load management (i.e. scheduling) are two orthogonal problems that should be treated separately. They propose a library that relies on a novel algorithm, known as *load control*, that regulates the number of active threads depending on the load. Basically, to acquire a lock, a thread either (1) blocks, if there are too many threads running or (2) spins otherwise. That way, threads are very reactive since they use spinlocks as much as they can and performance never collapses because the load is regulated.

To limit the load, threads are only allowed to run if they are unable to register themselves into a bounded array known as the *sleep slot buffer* because it is full. More precisely, the proposed algorithm—implemented by the load control library—works as follows: to acquire a lock, a thread spins until a) the lock is released or b) it is able to register itself into the sleep slot buffer. In the latter case, the thread blocks until it is removed from the sleep slot buffer or until 100ms pass, whichever comes first. Once the thread wakes up it restarts the lock acquire process as if it just arrived.

To control the number of running threads, a *controller* increases or decreases the size of the sleep slot buffer and wakes up threads when they get kicked out of the buffer. Thus, the controller acts as a local scheduler that only manages the load whereas the threads themselves only manage contention: contention and load management are effectively decoupled.

Experiments show that on various benchmarks, load control performs just as well as spinlocks under low load. Performance gracefully degrades under high load instead of collapsing as is the case with spinlock-based algorithms. Block-based algorithms' performance also degrades gracefully under high load, but they are initially much slower. Load control therefore offers the best of both worlds. It is worth noting that the spinlocks and blocking locks in the experiments used state-of-the art locking algorithms, which shows how efficient Johnson et al.'s approach is. Load balancing is orthogonal to our work, since it aims to improve locking by making locks load-aware, whereas we aim to improve locking by improving data locality.

2.3. Efficient inter-process communication

Another way to improve performance in multicore environments is to improve communication between processes. This issue has been mainly tackled by OS designers and researchers, since OSes have to handle multiple processes running on multiple cores. Two opposing approaches are

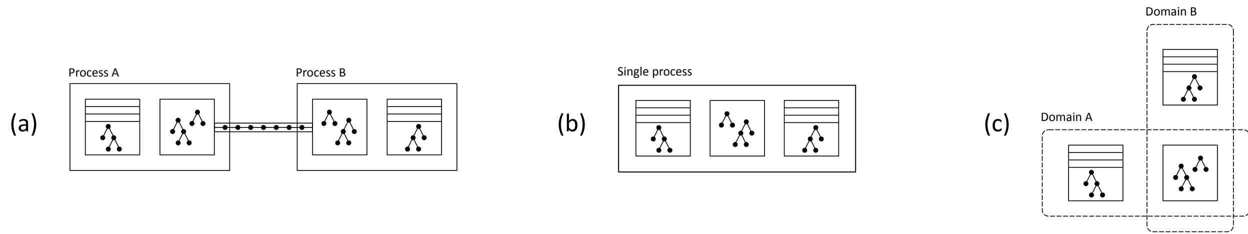


Figure 1: Two components sharing data (a) using standard, marshalled communication over pipes, (b) directly, both components being embedded into the same process, and (c) using Opal's approach with memory protection domains decoupled from execution domains (Source: [4]).

generally used for the communication between processes: shared memory and message-based communication.

This dichotomy is mainly an opposition between two isolation paradigms. Some (Chase et al. [4], for instance) argue that memory should be easily sharable. They design, for instance, OSes with global address spaces to improve sharing. Others argue (Baumann et al. [2]), Fähndrich et al. [6, 8]) that, on the contrary, shared memory should be prohibited by the OS. Both approaches have various repercussions on performance, safety, and ease of development. In this subsection, we focus on performance only, safety issues being beyond our scope. Performance-wise, the main difference between the two approaches is that when shared memory is directly used by applications, the OS cannot control the use of shared memory areas, which can lower performance. As for the ease of development, this matter is beyond the scope of this document since we only plan to change the inner workings of an JVM in our project, without having any direct influence on the development of higher-level applications on a semantic level.

2.3.1. Shared memory

Shared memory is the most common way by which processes communicate in traditional OSes. Its main advantages are simplicity and flexibility. However, sharing memory can cause contention problems that need to be overcome with safety and security mechanisms.

The Opal operating system, designed by Chase et al. [4], provide a good example of performance-optimized communication by shared memory. Opal uses a single 64 bit address space for all applications. This allows programs to directly use pointers to memory locations that they do not own, without the need for translation mechanisms between address spaces. Since sharing pointers to memory between programs with Opal is so easy, data is rarely ever copied from one program to another—transfers are all zero-copy. Traditional operating systems usually offer mechanisms for zero-copy communication, but they tend to be very complex. Another way to use easy zero-copy mechanisms between two components would be to place them in the same process, but this compromises protection and hinders modularity. Opal's decorrelates memory protection from execution, thereby allowing distinct processes to directly work on the same data. This approach, illustrated in Figure 1, combines simplicity with modularity.

In Opal, segments of memory can be shared across processes with protection mechanisms based on capabilities. In order to allow several applications to work together safely on in-memory data structures, data from a shared segment is accessed through shared procedures that are also stored in the same segment. This technique permits to maintain a high level of isolation and modularity while still benefiting from the very high performance provided by Opal's direct sharing of data.

Chase et al. show that Opal allows several applications from the Boeing CAD system that use the same data to directly work together on in-memory data structures without the need for marshalization, copy, or address-space translation. A custom benchmark is also used to evaluate Opal's performance. This benchmark uses three tools, (1) a producer that creates and deletes fixed-size records, (2) a consumer that maintains its own index structure on the same data and (3) a mediator that keeps the index up to date between the consumer and the producer. Experiments with this benchmark show that using directly shared segments (i.e. without address translation) is as efficient as embedding the three tools into the same process, and up to ten times more efficient than using three processes that communicate via traditional IPCs.

Opal's use of a global address space is therefore very efficient. However, it has several drawbacks, in particular, it uses an abstraction layer above the hardware that makes impossible for applications to use advanced, low-level optimizations on complex architectures that feature a large number of cores.

2.3.2. Message-based communication

Recent works on OSes optimized for multicore architectures show a tendency to discard shared memory altogether and to vouch for communication via messages only instead. Indeed, communication via shared memory has several drawbacks, in particular, on architectures with a large number of cores, using shared memory presupposes cache coherency mechanisms. These mechanisms are increasingly complex to implement and their overhead is getting worse as the number ef cores increases. Some recent processors do not ensure coherence between caches. Furthermore, at a lower level, these mechanisms are implemented by message passing. Allowing applications to directly handle the message passing mechanisms allow them to send and receive just as many messages as they need, without the need to assert the strong invariant of cache coherence, thereby improving performance.

We are now going to discuss two recent research OSes for multi-core (or multi-processor) architectures that make use of message-passing as their sole means of communication: Barrelfish (a multikernel) and Singularity.

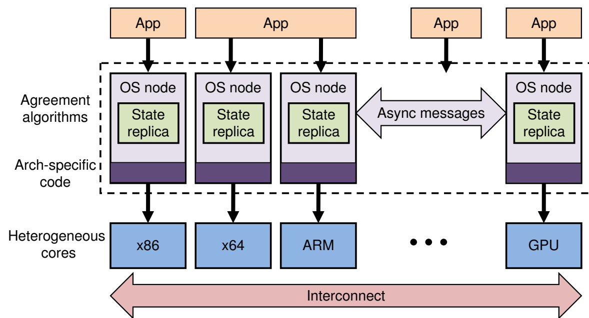


Figure 2: General architecture of Barrelfish (Source: [2]).

Barrelfish

Baumann et al. [2] argue that designing OSes that handle better architectures having a large number of cores[1] is a crucial research goal. Indeed, commodity computer systems containing multiple cores and/or processors are becoming more and more common. Most mainstream OSes have historically been developed for monoprocessors or multiprocessors having a limited number of processing units, using a model that scaled well in this context (global structures with locks, communication via shared memory, etc.). This model however has difficulties scaling well with newer hardware due to contention. Given the increasing number of cores in modern architectures and the complexity of interconnections between processing units, computer systems resemble more and more medium-scale distributed systems: Baumann et al. argue that designing an OS like an actual distributed system could improve scalability.

A Multikernel is a new type of operating system proposed by Baumann et al. that aims to this problem. Multikernels are multithreaded, with an instance of the OS running on each available core. Cores do not share global structures as in a traditional OS; instead, the OS state is replicated on each one of them. Cores do not communicate via shared memory, relying on cache coherence algorithms; instead, all communication is explicit and happens through asynchronous messaging.

Baumann et al. wrote an experimental version of a Multikernel named Barrelfish. In Barrelfish, a *CPU driver* and a *monitor* are bound to each core. CPU drivers handle system calls, schedule processes and threads on their cores and perform other low-level core-bound operations. Monitors are processes running on each core that coordinate the system-wide state via messages and encapsulate most of the higher-level functions that are usually found in a traditional kernel. The architecture of Barrelfish is shown in Figure 2.

The ability to send messages from one process to another within the same core or across cores is provided to user applications. However, applications can also communicate through shared memory like in a regular OS.

Baumann et al.'s experiments evaluate the performance of Barrelfish's basic features. They focus on the OS' handling of concurrency, messaging, computation and I/Os.

[1] This is reminiscent of Corey. Indeed, Corey, Barrelfish and as we will see later, Singularity, all constitute recent attempts at dealing with this problem.

Through these experiments, Baumann et al. show that the performance of their messaging facilities is comparable to L4's IPCs, which is rather good even though L4 is not a fully-featured, widly used microkernel. Performing a TLB shootdown (i.e. invalidating pages in the Translation Lookaside Buffer) and therefore unmapping pages is much faster on Barrelfish than on traditional OSes when the underlying hardware uses more than 14 cores on their test configuration, showing Barrelfish's greater scalability in this context. They also show that Barrelfish is faster than Linux for sending and receiving messages through the IP loopback (these tasks involve the messaging, buffering and networking subsystems of the OS). Other experiments show that Barrelfish has a comparable performance to that of Linux for compute-bound and IO workloads.

Even though Barrelfish is an experimental OS, its performance rivals that of classical, highly-optimized OSes on computer systems that have a large number of cores, which tends to show that the Multikernel approach effectively delivers in terms of scalability. Let us note, however, that traditional OSes may have not been optimized for these systems yet though, so we cannot be sure they will not scale well.

We have now seen Baumann et al.'s take on the use of message-passing only for communication between processes and threads. Barrelfish is not the only such operating system that has been designed, however. Fähndrich et al.'s Singularity [6] (and Helios [12], which is based on Singularity) also explored this path: we will focus on this operating system in the next subsection.

Singularity

Fähndrich et al. also propose to design an operating system that uses message-passing as their sole means of communication. It is worth noting that their main motivation for building such an OS is not performance, but modularity and security. In the current subsection, we mostly focus on performance issues.

Singularity [6] aims to overcome a major drawback of message-based communication: data is often copied instead of shared when using the event-driven paradigm, which leads to poor performance. To overcome this, Singularity uses a new programming language based named Sing# that provides efficient messaging capabilities.

Messages are exchanged over bi-directional channels. These channels can also be used to exchange channel endpoints or pointers to memory location. This last point is crucial: it allows for fast data transfers by removing the burden of copying memory blocks.

In Singularity, processes are isolated and individually garbage-collected. Therefore, passing data from one thread to another could be an obstacle to safe garbage collection. Sending data pointers to allow for zero-copy messaging seems like an even more complex task: one could wonder which garbage collectors are responsible for such pointers. Singularity uses strong ownership invariants to solve this problem.

Let us consider this approach more in detail. When a message is sent, references to message arguments or transfered data blocks can be found in both the sending and receiving threads, for instance. To solve this issue, the Sing# compiler statically checks that processes only access memory that they own and that a memory block al-

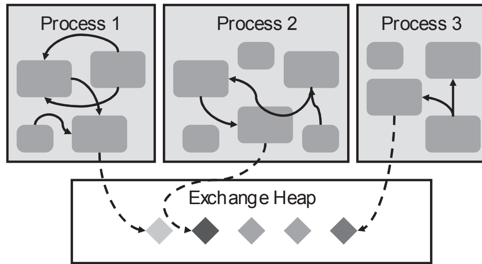


Figure 3: Singularity's exchange heap (Source: [6]).

ways belongs to a single thread at any given time. To this end, data on the GC heap is separated from data in the *exchange heap*, as illustrated on Figure 3. The exchange heap is used to exchange all data between processes, i.e. messages and memory pointed to by exchanged pointers. At any time, each memory block is owned by a function in a process: ownership is automatically passed from a function (and thread) to another through simple rules—when a variable is passed as a parameter, its ownership is passed to the called function for the whole time of its execution, for example. The only exception to this automatic ownership management strategy is when memory pointers are exchanged between threads: to explicitly track memory blocks from the exchange heap whose pointers are exchanged *via* messages, a special type of object, *TCell*, is provided. Sing#'s strong type-checking mechanisms combined with this system of ownership management allows the whole system to dismiss hardware memory protection altogether, which removes a major cause of overhead.

Performance experiments show that Singularity is faster than Linux and Windows for certain tasks. Performing a system call, switching between two threads, and, more importantly, sending a message are faster on Singularity. Moreover, the authors show that sending a block of memory from one thread to another is fast and does not depend on the size of the block. Linux and Windows are slower at performing this task, especially for larger blocks, since they copy the data instead of just passing a pointer.

This analysis of Singularity and Barrelfish showed that inter-core communication based on specialized algorithms improves data transfer performance. These algorithms will be a major source of inspiration to us since we will need to design efficient communication algorithms to transfer control and variables between cores when migrating synchronized blocks. Moreover, the works of Baumann et al. and Fähndrich et al. showed that limiting the use of shared memory could improve performance by limiting the underlying work of cache coherency algorithms, which is the main basis of our project: we wish to delegate synchronized blocks to dedicated cores in order to limit the overhead caused by the transfer of cache lines between cores.

In this subsection, we discussed solutions to transfer data from one thread to another. We will now focus on a related problem: the transfer of control between threads.

2.4. Fast transfer of control

Optimizing the transfer of data from one thread to another is a good way to improve performance on multicore architectures, since it is linked with the major problem of these architectures: contention. However, another way to improve performance is to speed up the transfer of control from one thread to another, i.e. calling code from another thread. Indeed, we are in dire need of efficient mechanisms to dispatch the execution of code to threads: our project consists in finding efficient ways to transfer the control from one thread to another to delegate the execution of synchronized blocs.

Calling remote procedures from one thread to another is usually done through Remote Procedure Calls, or RPC. RPCs usually permit to call remote procedures from one core to another, as well as from a computer to another on a network. Such calls encapsulate the underlying transmission mechanisms, allowing developers to call procedures transparently without having to worry whether calls are local (between threads) or remote (over a network). Such an abstraction is usually viewed as a good thing, since it permits to write distributed systems on a network with very simple semantics. We are not interested in such capabilities, however: we only wish to dispatch the execution of code between cores in an efficient way.

In this subsection, we focus on three works that provide solutions to these issues : Bershad, et al.'s URPC [3], Huang et al.'s LRPC [7] and Ford et al.'s proposition of a migrating thread model [5].

URPC

URPC [3] aims to facilitate the design of highly-specialized RPC systems. Using custom RPC systems instead of generic ones makes it possible to use semantics that are not always provided by standard RPC packages. The loss of genericity also permits to improve performance, since unused parts of the RPC mechanism can be removed, thereby removing unnecessary overhead (messages, CPU time).

URPC comprises a runtime library, a stub generator and an name server, all of which are highly customizable. The URPC runtime library (see shaded boxes in Figure 4) makes it possible to customize the *protocol machines* of the RPC mechanism (*Client PM* on Figure 4) as well as *communication services*. The protocol machines encapsulate the RPC topology, call semantics and failure semantics of the RPC system. The protocol machine can be customized using *Cicero*, a protocol description language that handles advanced features such as multithreading. On the client side, the protocol machine's routine for assembling and disassembling messages (*Client RPC*) is factored out of the protocol machine. This routine allows, for instance, to customize marshalling and unmarshalling functions. The communication services component comprises routines to facilitate point-to-point communication between RPC participants, using a generic send/receive model that makes it possible to model any type of communication between nodes.

The stub generator includes special annotations to RPC signatures to support advanced communication schemes. These extensions allow for peer-to-peer communication (as well as standard client/server communication) and communication schemes that require multiple calls to complete for a given callback, such as asynchronous communication.

The name server uses a generic naming structure that accommodates for different naming models (a simple colon-separated list of attributes). It handles simple and

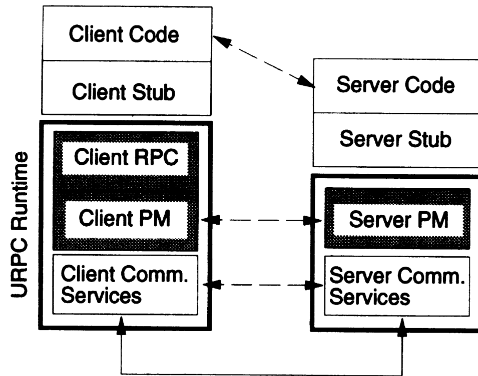


Figure 4: Architecture of the URPC runtime library (Source: [3]).

multiple end-point lookups to support unicast as well as multicast communication. In order to locate sets of hosts corresponding to given characteristics, numerical boolean relationships (such as $<$, $>$ or $\neq$, for instance) can be used to select hosts based on their attributes.

Bershad et al. show through examples that URPC can be used to design various types of RPCs, such as multicast RPCs, asynchronous RPCs and callback RPCs. These two last possibilities show that URPC is suitable for our project. Furthermore, their implementation of a special type of RPC that use at-most-once failure semantics shows a 10% performance gain over SUN RPCs, which shows that custom RPCs can be at least as fast as normal RPC.

On a final note, Bershad et al. advertise URPC as a tool to prototype RPC systems. However Barrelfish uses URPC as its main communication mechanism, which shows that URPC can be used in complex projects where performance is an issue.

LRPC

LRPC [7] (for *Lightweight Remote Procedure Call*) is a RPC library optimized for communication between cores on the same machine. This is particularly interesting to us since we do not need to transfer control to remote machines in our project.

With LRPC, calls to procedures are implemented with kernel traps. When a client starts a RPC, a kernel trap is called. The kernel validates the caller, creates a call linkage, and dispatches the client's thread to the server process. The client provides the server with a pointer to an argument stack as well as it thread of execution. The kernel switches to this thread with an upcall. After the procedure call completes, control and results return through the kernel back to the point of the client's call.

Three main optimizations are used. First, the kernel switches threads by updating a few registers (the stack pointer, the virtual memory registers, and a few other ones to pass the information described above) and by performing an upcall. This is much more efficient than performing a true context switch. Second, arguments are only copied to argument stacks (in shared memory) then back into the client thread, instead of needing up to seven copies in with regular RPCs[2]. Third, on multicores architectures, server processes are cached on idle processors. Thus, on a procedure call, if the server process is available on an idle core, the RPC can be processed very quickly, since a context switch is not needed. It is also worth noting that, according to Huang et al., the multiprocessor implementation avoids contention on data structures as much as possible.

LRPC is 2 to 3 times faster than Taos[3] RPC on a simple benchmark performing 100,000 RPC requests in a loop. On four processors, LRPC is 3.7 times faster than on a single processor with the same benchmark. This shows how interesting LRPC is for our project: it provides fast transfer of control capabilities on multiprocessors.

Migrating thread model

Ford et al. propose another compelling way to speed up transfer of control between threads: the *migrating thread model* [5]. In traditional OSes, threads belong to processes and cannot move from one process to another. With the migrating thread model, a thread abstraction moves between processes with the logical flow of control.

To manage this, Ford et al. decouple the *execution context* from the *schedulable thread of control*. The execution context encapsulates the state of the registers, program counter, stack pointer, and references to the containing process and designated exception handler, whereas the schedulable thread of control represents the thread itself in which the program is executed.

Ford et al. implemented their migrating thread model above Mach 3.0's kernel. In pratice, threads are based into the kernel, and make incursions into processes (in user mode) via upcalls. Moving a thread from one process to another is cheaper than a context switch: the kernel only reproduces part of the functionality implemented by the scheduler. For instance, since the kernel is now calling the server rather than vice-versa, it no longer needs to save and restore the server's registers on every RPC. Moreover, thanks to the server's first-hand knowledge of the RPCs, it no longer needs to create, translate, and consume reply ports[4] to match a reply to its request. Also, the kernel no longer needs to manage message buffers since the data is directly copied from source to destination.

Migrating threads are especially powerful when combined with RPCs, allowing for fast transfer of control on a single machine. Ford et al. implemented their thread model on the Mach 3.0 kernel. They show that regular RPCs require 5 times as many instructions and 4 times as many load/store operations than RPCs that use the migrating thread model.

This ends our subsection about fast transfer of control and our discussion of optimizations for homogeneous architectures. Indeed, until now, we discussed possible performance optimizations based on the hypothesis that the target architecture's cores share the same performance, ISAs, and characteristics. We will now extend our discussion to heterogeneous architectures.

[2]The following copies are usually needed with unoptimized RPCs: from the client thread to the message, from the sender domain to the kernel domain, from the kernel domain to the receiver domain, from the message to the server stack, from the receiver domain to the kernel domain, from the kernel domain to the sender domain, from the message into the client result.

[3]Taos is the operating system developed for the for the DEC SRC Firefly multiprocessor workstation [14].

[4]A port is a protected message queue for communication between tasks in the Mach kernel.

2.5. Heterogeneous architectures

Heterogeneous architectures, i.e. architectures whose processing cores differ in computing power, ISAs, and other characteristics, are getting more and more common. Even standard PCs can be viewed as heterogeneous architectures, since several of their components (NICs, GPUs) can be used as independent processing cores. Being able to optimize the performance of our approach (i.e. migrating critical sections to dedicated cores) on heterogeneous architectures could be an interesting long-term goal. This is the object of this subsection.

Bias scheduling

Koufaty et al. propose a scheduling algorithm that aims to improve performance on a common case of heterogeneous architectures: those having two types of cores only (faster and slower ones, called *big* and *small* cores, respectively). Their algorithm is novel in that it doesn't require offline profiling or dynamic sampling on all cores: it collects all of its data on-the-fly.

Koufaty et al. propose new metrics to evaluate the performance gain of executing a process on a big core rather than a small core (i.e. *application bias*). We mentioned in Section 2.1 that in addition to the traditional *Clocks Per Instruction* (CPI) metric, they also monitor internal stalls (caused by accesses to resources internal to the core) and external stalls (caused by accesses to shared last level caches, memory and I/O). They show that a process has a *big core bias* (i.e. its speedup from running on a big core compared to a small core is large) when the number of stalls (especially external stalls) is low. The number of internal and external stalls is measured at runtime and used to compute application bias. As this bias is not constant during the execution of an application, it is measured over a sliding instruction window. This allows the algorithm to schedule more efficiently applications whose behavior changes throughout their execution.

When the system is unbalanced, the scheduler tries to migrate threads from the busiest core to the idlest core, using application biases to optimize its choice of threads to move. When the system is balanced, the runqueues of each core are periodically checked: the scheduler looks for processes that would benefit from being swapped from big to small cores and conversely according to their application bias.

Since heterogeneous chips are uncommon, Koufaty et al. emulate one via an homogeneous quad-core Intel Xeon processor. They show that the standard approach of downclocking some of the cores to turn them into small cores is not representative of architectures that have structurally diverse cores. This is an issue because these architectures are expected to be the obvious choice for manufacturers, since it makes it possible to optimize the smaller cores for physical size and energy efficiency. Therefore, they use proprietary tools to enable a debug mode on some cores that reduces instruction retirement from four to one micro-op per cycle. They show that this throttling method gives performance results similar to actual heterogeneous architectures whose small cores are in-order whereas their big cores are out-of-order.

To evaluate performance, Koufaty et al. perform experiments with heterogeneous workloads (i.e. workloads whose processes have very different biases) based on the SPEC CPU2006 benchmark. On average, they obtain a 9% performance improvement over the default Linux scheduler. This is close to an upper bound found by running the tests on the same processor without throttling any of the cores. They also perform experiments with more homogeneous workloads and obtain a 5% gain on average. They show that this improvement is due to the fact that even though the application biases are similar overall, they vary throughout the execution of each benchmark.

Koufaty et al.'s work is extremely interesting to us, since it permits to improve the scheduling of applications on a common case heterogeneous architectures: performance asymmetric CPUs. Other research works have focused on more general heterogeneous architectures, considering each processing unit in a system (such as NICs or GPUs) as a normal CPU on which processes can be executed.

Multikernels

We discussed multikernels in Section 2.3.2. Multikernels have been designed with such heterogeneous architectures in mind. Indeed, Baumann et al. argue that architectures are getting increasingly diverse (memory hierarchies, instruction sets, interconnects, etc.) and that OSes have been statically optimized for the most common architectures at a low level for now. Given the varying nature of workloads and the diversity of hardware designs in modern computer systems, they argue that this approach will likely not be efficient enough anymore in the near future. Baumann et al. propose that designing OSes like distributed computer systems allows them to adapt better to various architectures, just like network applications are able to dynamically adapt to architecturally diverse networks.

Multikernels are mostly hardware-independent. The only architecture-specific modules are the messaging transport mechanism and the interfaces to the hardware (CPUs/cores and devices). Bauman et al.'s implementation of a multikernel, Barrelfish, uses a *knowledge database* containing information regarding the underlying hardware (found by polling and measurements) that it uses to optimize its communication scheme. This database can be used to select appropriate message transports for inter-core communication or to allow for NUMA-aware memory allocation, for instance.

Helios

Helios [12], an OS based on Singularity (cf. Section 2.3.2), provides an interesting alternative solution to the problem of handling heterogeneous architectures. Helios does not rely on multiple identical kernels, instead, it uses a main kernel and satellite kernels that (1) export a standardized set of kernel abstractions on all cores, (2) provide transparent and unified IPCs[5] for communication between cores, and (3) support heterogeneous ISAs via an intermediate bytecode to which applications are compiled.

Helios uses an *affinity metric* to automatically place applications on cores. This affinity metric allows developers to express the fact that two processes would benefit from running on the same core (or the opposite). This is rem-

[5] Helios' IPCs uses Singularity's zero-copy mechanism for local message parsing if used between two local threads. Otherwise, Helios transparently marshals messages between the initiating thread and the remote service.

Name	Layer	Communication scheme	Contention	IPCs/Tr. of control	Heterogeneity	Performance	Scalability	Safety	Customizability
Multikernel	OS	URPC	+		+		+	+	
Singularity	OS	LMP (local, zero-copy)	+	+			+	+	
Helios	OS	LMP + RMP (distant, marshalling)			+				
Corey	OS	Shared memory	+			+	+		
Opal	OS	Shared memory (global address space)		+		+			
Bias scheduling	library	n/a			+	+			
URPC	library	URPC		+		+			+
LRPC	library	LRPC	+	+		+	+		
Migrating threads	library	Migrating threads		+		+			
Load scheduling	library	n/a	+			+	+		

Figure 5: Summary of the improvements from the research works analyzed. '+' is a shorthand for "substantial improvement". VMKit and DProf are not listed because the chosen criteria were not applicable.

iniscent of the implementation of bias scheduling that we discussed earlier.

We will not elaborate any further on the mechanisms provided by operating systems such as Multikernel and Helios to handle complex heterogeneous architectures since this subject is beyond the scope of our project for now.

2.6. Conclusion

In this section, we have reviewed a selection of scientific publications that tackle best the issues we are faced with in our project, at least to our knowledge. Figure 5 summarizes the improvements provided by each research work we mentioned our analysis. The next section describes the experiments we performed on thread migration to ensure that our approach is viable.

3. Experiments

For our experiments, we used two machines, featuring 8 and 48 cores respectively. Since we only received the second (48-core) machine recently, data from this machine is still incomplete; most experiments have been performed on the 8-core machine only. These two machines are described below.

- The first machine, codenamed *bossa*, is a Mac Pro (3.1) featuring two X5472 quad-core 3.2Ghz Xeon CPUs. Each of the two processors combines two dual-core dies[6]. Bossa uses a Intel 5400 chipset with a 1.6Ghz system bus and 10GB of DDR2 SDRAM. It runs Debian Linux 5.0.5 with a 64-bit 2.6.31 Linux kernel.
- The second machine, *amd48*, uses four "Magny-Cours" 6172 Opterons, featuring two six-core dies. The four CPUs are interconnected via HyperTransport links. Amd48 uses an AMD RD890 chipset with 32GB of DDR3-SDRAM. It runs Mandriva Linux 2010.1 (x64) with a 2.6.34 Linux kernel.

Our aim is to determine the parameter spaces under which each of these two solutions are the more efficient:

- **Solution (a):** execute the critical sections from within the thread they belong to. The main overhead of this solution is caused by cache misses when shared variables are not available in the L1 cache and have to be fetched from remote cores (or higher-level caches).
- **Solution (b):** execute the critical sections on a dedicated core[7]. This solution induces two major causes of overhead: (1) it requires a transfer of control from the local core to the dedicated core (modelized by a local RPC in our benchmarks) each time a critical section is executed and (2) since local context variables are not directly available on the distant core, they have to be transferred, similarly to shared variables in solution a.

The simplest way to modelize this issue is to consider that Solution (b) is more efficient than Solution (a) when:

Equation (A): $c_{rpc} + n_{context}m < n_{shared}m$

Where c_{rpc} is the cost of a transfer of control (i.e. a RPC between cores), $n_{context}$ and n_{shared} are the number of context and shared cache lines respectively, and m is the average cost of fetching a cache line from a distant core. This is a simplification, of course, because m varies depending on where the variable is available in the cache hierarchy. Both $n_{context}$ and n_{shared} are dependent on the programs being executed in the Java Virtual Machine and cannot be evaluated precisely. On a given architecture, however, c_{rpc}, and m can be measured. If the values of these three variables are known, we can obtain a gross estimate of the parameter space, i.e. the values of $n_{context}$ and n_{shared}, in which Solution (b) is more efficient than Solution (a). This is the object of this section.

In Section 3.1, we evaluate the cost of cache misses, in Section 3.2 we evaluate the cost of a transfer of control, and in Section 3.3, we provide a rough estimate of the values of $n_{context}$ and n_{shared} for which Equation (A) is verified.

[6] In the context of CPUs, a die is a small block of semiconducting material on which one or several cores along with other components are fabricated. Communication between dies is slower than communication between cores on the same die. Cores on the same die usually share a cache chip (typically, the L2 cache).

[7] We ignore the case where several cores are dedicated to the execution of critical sections for now, to simplify the analysis.

3.1. Rough estimates

This subsection evaluate the cost of m and c, i.e. the average cost of a cache miss and the cost of a transfert of control.

3.1.1. Cost of cache misses

In order to evaluate the cost of cache misses, we used *memal*, a program from the benchmark suite used by Wickizer et al. to evaluate the performance of the Corey operating system [15]. Memal uses two threads, a server and a client thread, each of these threads being pinned on two different cores specified by the user. A memory area (C array) is shared between the two threads. The server thread uses low-level techniques to place the memory area within its L1 cache, L2 cache or an owned memory area. The client thread then reads each cache line from this buffer, and the benchmark returns the average number of cycles needed to read each cache line.

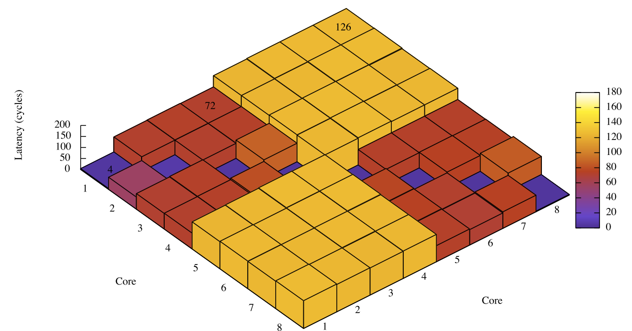
Memal uses low-level techniques initially proposed by Yotov et al. [16] in order to prevent the compiler from removing data accesses and to prevent cache prefetching from altering the results. More precisely, the shared memory area is not accessed by the client in a linear fashion. Instead, each 64-bytes word of this memory area contains an address to another word of the same memory area, in such a way that by reading the first address, using the read address to know which address to read next, and so on until enough 64-byte words have been read, one can read the whole memory block in a pseudo-random fashion. In the end, the last value is passed to a system call that does not produce any effect (a `printf` with an empty string as its first parameter). Thus, the reads are dependant from each other, and the last read value is used: this prevents the compiler from thinking some of the memory accesses can be removed as an optimization. Moreover, since 64-byte words are accessed randomly, the processor cannot efficiently prefetch cache lines. This technique might prove important for us in some of the benchmarks we are about to write, even though for now we found simpler–albeit platform-dependent–workarounds to avoid optimization and prefetching issues.

The results of the memal benchmark on bossa are shown in Figure 6. Each graph shows the number of cycles needed for a client on Core i to access data located on Core j, in the L1 cache, L2 cache and in the RAM, respectively. The core IDs of the client and server threads are on the X and Y axes of each graph[8].

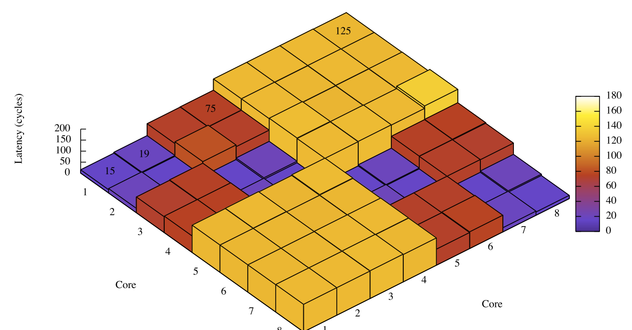
Bossa uses two Xeon X5472 processors, each of which has two dual-core dies. Each core has a local L1 cache, and the L2 cache is shared between each two cores from the same die. There is no L3 cache, the third level is the RAM itself. Since Bossa uses NUMA (Non-Uniform Memory Access), access times to the RAM are processor-dependent.

The first graph exhibits three different latency access times to the L1 cache. accessing the local L1 cache takes

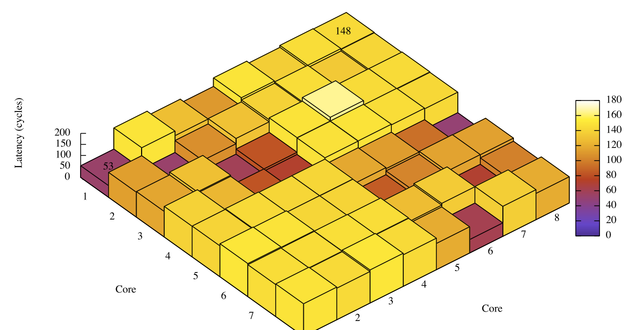
[8]Since all graphs are symmetric, it does not matter which axis represents client and server threads: it takes as much time for a client thread on Core i to access a variable in a given cache level (or in the RAM) from Core j than for a client thread on Core j to access a variable in a given cache level (or in the RAM) from Core i.



Latency of L1 cache accesses



Latency of L2 cache accesses



Latency of RAM accesses

Figure 6: Latencies of cache RAM accesses relative to the cores between which cache lines are transferred, on *bossa*, as measured with the *memal* benchmark. Each processor uses two Dies, each die use two cores. Cores 1 to 4 belong to the first processor (Die 1: Core 1 and 2; Die 2: Core 3 and 4), and Cores 5 to 8 belong to the second processor (Die 1: Core 5 and 6; Die 2: Core 7 and 8).

Placement of the client and the server	Number of cycles	Number of L1 cache misses	Number of L2 cache misses
Same die	∼302	1	0
Same processor, different die	∼412	1	1
Different processors	∼968	1	1

Results of the `null_rpc` benchmark with both synchronization variables on the same cache line

Placement of the client and the server	Number of cycles	Number of L1 cache misses	Number of L2 cache misses
Same die	∼547	2	0
Same processor, different die	∼525	2	2
Different processors	∼1190	2	2

Results of the `null_rpc` benchmark with both synchronization variables on two different cache lines

Figure 7: Results of the `null_rpc` benchmark on bossa. Note: the number of cache misses was measured on the server only, therefore, the total number of cache misses is the double of the number given.

```
1 for (i = 0; i < number_of_rpcs; i++)
2 {
3   rpc_requested = 1;
4
5   while (!done_with_rpc)
6     pause;
7
8   /* Protected code */
9
10  done_with_rpc = 0;
11 }
```

Critical section from the client thread

```
1 for (i = 0; i < number_of_rpcs; i++)
2 {
3   while (!rpc_requested)
4     pause;
5
6   /* Protected code */
7
8   rpc_requested = 0;
9   done_with_rpc = 1;
10 }
```

Critical section from the server thread

Figure 8: Pseudo-code of the critical sections from the client and sever thread in the `null_rpc.c` benchmark.

about 5 cycles. Accessing a variable from the L1 cache of any other core on the same processor takes about 75 cycles, because in each case the variable has to be fetched by the L2 cache, then passed back to the local L1 cache. When accessing data located on the L1 cache of a remote processor, 125 cycles are needed. The second graph shows that when the data is available from the L2 cache, the access times are 15-20 cycles. When the data has to be fetched from a local L2 cache, the latency is about 75 cycles. Fetching data located in the cache of a remote processor takes about 125 cycles. Finally, the third graph shows that the access time to a variable from the RAM itself is highly variable if done locally (i.e. on a single processor), with access times varying between 50 and 140 cycles. Accessing data from the RAM takes about 150 cycles if the variable has been modified by the remote processor. This difference of latency between variables accessed locally (i.e. same processor) and remotely (i.e. different processor) shows that because of NUMA, access times to the RAM are not constant.

3.1.2. Cost of transfers of control

To evaluate the cost of transferring the control between two processors, we implemented a simple benchmark that uses two threads, a client and a server, pinned on cores whose IDs are provided by the user. In this benchmark, the clients request the executions of a given number of RPCs by the server. The benchmark uses the PAPI library [1] in order to measure the number of cycles per RPC, or the number of L1/L2 cache misses. The code of this benchmark (`null_rpc.c`) is available in the appendix.

RPCs are not implemented using blocking locks, instead, we used spinlocks, in order to minimize the reaction time. We are of course wasting cycles (cf. Section 2.2.2), but in a system having a large number of cores, one could consider that wasting cycles from a number of unused cores is negligible. Minimizing the cost of RPCs is crucial however, since we hope the performance gain from increased data locality will be greater than the loss in performance caused by RPCs. The full code of this benchmark is provided in the appendix (`null_rpc.c`).

Figure 8 shows the pseudo-code of the critical sections of the client and the server threads. For each iteration, the client thread sets the `rpc_requested` variable to 1. This automatically starts a RPC. Then, the client busy-waits[9] for the server to set the `done_with_rpc` variable to 1. When this happens, the server is done with the RPC. The client sets its `done_with_rpc` variable to 0 and starts the process again. On the server side, for each iteration, the server busy-waits for the `rpc_requested` variable to be set to 1. When this happens, the client has requested a RPC. The server just resets the `rpc_requested` variable to 0 and sets the `done_with_rpc` variable to 1, to warn the client that it is done with processing the RPC.

We ran this benchmark on bossa using two different configurations. In the first configuration, the two synchronization variables, `rpc_requested` and `done_with_rpc` are

[9]The pause x86 instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance[9].

```
1  for (i = 0; i < number_of_rpcs; i++)
2  {
3    *local_sv = 0;
4
5    while (CAS(global_sv,
6             0,
7             &local_sv) == 0)
8      pause;
9
10   /* Protected code */
11
12   while (*local_sv == 0)
13     pause;
14 }
```

Critical section from the client thread

```
1  for (i = 0; i < number_of_rpcs; i++)
2  {
3    while (*global_sv != 0)
4      pause;
5
6    /* Protected code */
7
8    **global_sv = 1;
9    *global_sv = 0;
10 }
```

Critical section from the server thread

Figure 9: Pseudo-code of the critical sections from the client and sever thread in the first version of the benchmark from Section 3.2.

on the same cache line, and in the second configuration they are on different cache lines. In the second configuration, we actually leave two full cache lines between the two variables, because on our architecture, when a cache line is fetched, the next one is automatically prefetched. The results are presented in Figure 7.

Figure 7 shows that if the client and the server are on the same core with synchronization variables on the same cache line (resp. on different cache lines), each RPC takes about 302 (resp. 547) cycles; with two cores on the same processor but on different dies, each RPC takes about 412 (resp. 525) cycles, and with two cores on different processors, each RPC takes about 968 (resp. 1190) cycles.

3.1.3. Cost comparison

Now that we have latencies for all cache levels and an estimate of the cost of RPCs between cores, we can estimate, in a restricted case (i.e. one client only), when our approach (Solution (b)) is faster than the traditional approach (Solution (a)).

- When the client and the server are on the same die, the minimum cost of a transfer of control is about 300 cycles, and the cost of transferring a variable from one core to another is between 15 and 20 cycles. With 20 more cache lines used by shared variables than cache lines used by context variables, Equation (A) is verified.
- When the client on the server are on the same core, but on a different die, the cost of a transfer of control is about 500 cycles, and the cost of transferring a variable from one core to another is about 75 cycles. With 7 more cache lines used by shared variables than cache lines used by context variables, Equation (A) is verified.
- When the client and the server are on different processors, the cost of a transfer of control is about 1000 cycles, and the cost of transferring a variable from one core to another is between 125 and 150 cycles. With 8 more cache lines used by shared variables than cache lines used by context variables, Equation (A) is be verified.

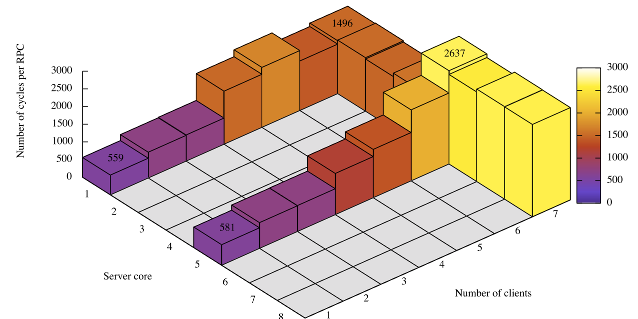


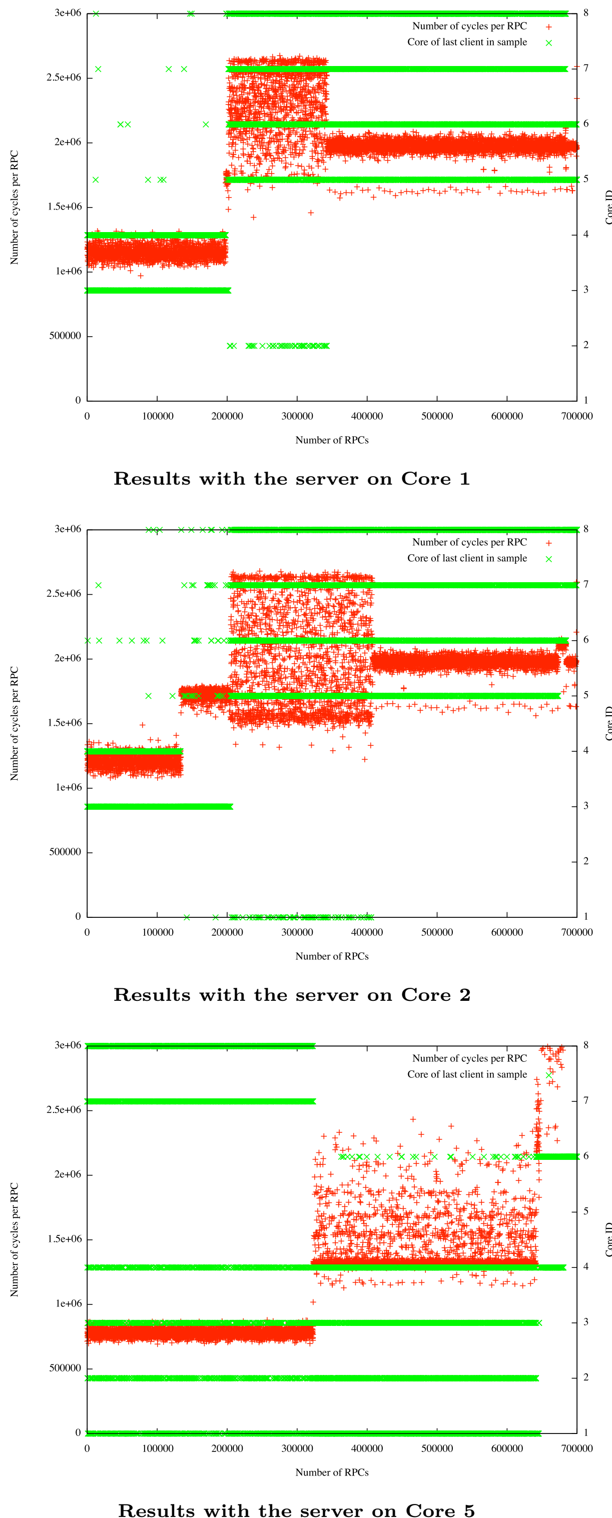
Figure 10: Number of elapsed cycles per RPC on bossa, depending on the number of clients and the Core n on which the server thread is pinned. The clients are pinned to Cores $n+1 \bmod 8$ to $n+c \bmod 8$ where c is the number of clients.

These initial results show that Equation (A) can be verified for a relatively low number of cache lines (8 to 20). Of course, the considerations from this subsection are not based on a microbenchmark that emulates an environment with multiple clients and with actual context and shared variables. Instead, since only one client was used, the transfers of the cache lines containing synchronization variables were limited and contention was inexistent. These first results are therefore very optimistic. Finding more accurate results with a more realistic microbenchmark is the object of the next sections.

3.2. NULL RPCs with multiple clients

The second phase of our work was to improve the initial `null_rpc.c` benchmark to use several clients (one client par core) in order to better simulate Solution (b). We wrote a first prototype whose critical sections are shown in Figure 9. The prototype uses $c+1$ synchronization variables where c is the total number of clients. Each client has a `local_sv` synchronization variable, and the server has its own synchronization variable, named `global_sv`. We do not use regular, blocking locks to avoid slow reaction times caused by context switches. Instead, we use a Compare-And-Swap (CAS)[10] primitive to optimize re-

[10] The *compare-and-swap* primitive executes the comparison and the 'swapping' of a variable atomically: first it compares



Results with the server on Core 1

Results with the server on Core 2

Results with the server on Core 5

Figure 11: Sampled executions on bossa (100000 RPCs per client, 100 RPCs per sample).

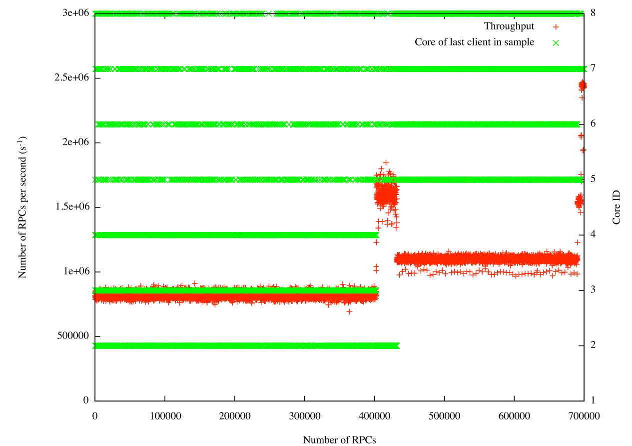


Figure 12: Sampled executions on Core 1 using a Test-And-Test-And-Set loop, on bossa.

sponse times. The general idea is that each client tries to set the `global_sv` synchronization variable to the address of its local `local_sv` variable using a CAS primitive in a busy-wait loop. When a client manages this operation, this means it managed to get the server to service its RPC. Since the server has the address of `local_sv` variable from the client whose RPC is being serviced, it can set it to 1 to indicate when it is done servicing the RPC. Since, we are only implementing null RPCs, the actual bodies of the RPCs on the server side is empty. The aim of the benchmark is to measure the elapsed time per RPCs, in number of cycles. Since we want all clients to execute their RPCs simultaneously, we use a synchronization barrier before entering the critical sections. The results of this first prototype on bossa are shown in Figure 10.

As shown in Figure 10, only a subset of all possible configurations were tested. First, we tried to use 1 to 7 clients with the server thread pinned to the first core of each processor and each client pinned to Cores $n+1 \bmod 8$ to $n+c \bmod 8$, where n is the core number of the server thread and c the number of clients. Then, for 7 clients, we ran our benchmark with the server on each of the 8 cores. Unsurprisingly, increasing the number of clients increases the latency, because of contention over the cache lines. More interestingly, the experiments give much better results when the server is located on the first processor. This clearly shows that the bus between the two processors is not symmetric.

Sampling

Even though we obtained coherent results with our first prototype, those are not very accurate because we noticed that small changes in the initialization code for the client and server threads caused the results to vary. For instance, adding a simple `i++` instruction before the synchronization barrier could, in some cases, add up to 500 cycles to the results with 7 clients (while preserving the same general tendency, i.e. that RPCs take less time to execute if the server is located on the first processor). To deal with this issue, we tried various approaches; in particular, we tried to service a single RPC par client and to save the order in which the clients were serviced, in order to understand whether the variance in execution times was correlated with this order. This solution proved unsuccessful.

the value of a variable with a given value, and if they are equal, the variable is set to another value given as a parameter. The compare-and-swap primitive returns a boolean to indicate whether the swapping happened.

```
1 while (CAS(global_sv,
2         0,
3         &local_sv) == 0)
4     pause;
```

TAS loop

```
1 for (;;)
2 {
3   if (global_sv == 0)
4   {
5     if (CAS(global_sv,
6         0,
7         &local_sv) == 0)
8       continue;
9
10      else break;
11  }
12
13   pause;
14 }
```

TATAS loop

Figure 13: TAS and TATAS loops from the benchmark prototype.

In order to better understand what was really happening during the experiment, we decided we needed to find a way to plot temporary results about the experiment in a temporal fashion. This lead us to divide the execution in samples.

Sampling the execution of the benchmark was a very simple idea that allowed us to better understand what happened at a lower level. The idea was to stop the execution of RPCs for a very short while every x RPCs to store data about the experiment. We chose to record (1) the number of cycles per RPC in each sample and (2) the core of the last client serviced in each sample. An example of result of such an execution is shown in Figure 11 (100,000 RPCs per client, 100 RPCs per sample).

Figure 11 shows that bossa's hardware does not ensure fairness, which is an issue we did not think of at first. On this figure, the red points show the average throughput during the previous sample (i.e. (1) from the previous paragraph) and the green points show the core of the last client serviced in each sample (i.e. (2) from the previous paragraph). The execution is split in phases, during which only some clients have their requests serviced by the server. Switches between phases happen when some of the clients are done executing their RPCs. For instance, let us consider the first graph in Figure 11. In this figure, the server is pinned to the first core of the first processor. The two clients located on the second die of this same processor are mainly serviced first, even if a few other clients get serviced occasionally[11]. This is Phase 1. When these two clients are done with their RPCs (both are done almost

[11] We only monitor the core of the last client in the sample in order to prevent the measurements from altering the result. In practice, that means that we only check which client gets serviced once in a while. This still shows a valid tendency regarding which cores get serviced, because the results clearly show that the execution works by phases during which only a subset of cores are serviced.

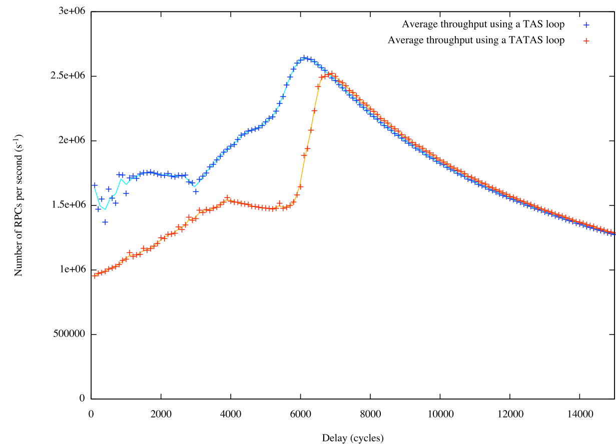
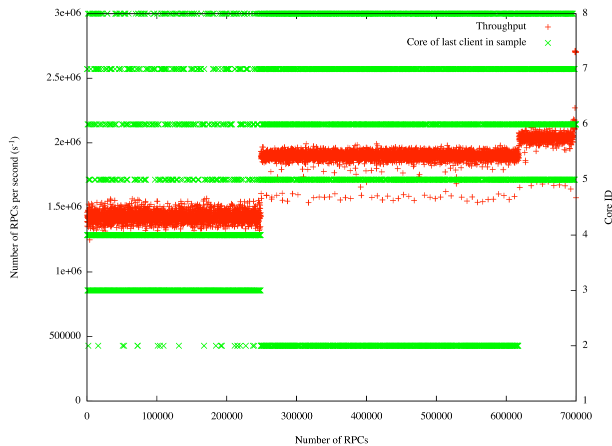


Figure 14: Average throughput as a function of the delay, using either a TAS or a TATAS loop, on bossa (7 clients).

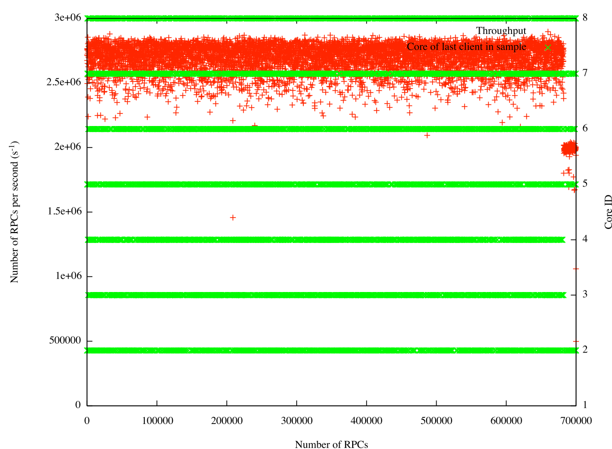
silmutaneously, which shows their RPCs were serviced at a similar rate) we enter Phase 2: all other clients get serviced, although the client from the second core (same die as the server) gets serviced faster than the others. Finally, when this client is done, the 4 clients from the remote processor are serviced at about the same rate. This is Phase 3. The value and variance of the elapsed time per RPC is characteristic of each phase: in the first phase, RPCs are serviced in about 2700 cycles, with a deviation of about 500 cycles. Then in Phase 2, RPCs are serviced faster : about 1200 cycles on average, with a greater variation, which is probably due to the fact that RPCs from clients located on the local processor are executed much faster than RPCs from clients on the remote processor. Finally, in phase 3, RPCs take more time to execute than in Phase 2 (because they all come from remote clients) but faster than in Phase 1 (because less clients try to execute RPCs). The deviation is very low because all RPCs come from the distant processor (same latency). The two other graphs from Figure 11 show executions from Core 2 (Processor 1, Die 1) and Core 5 (Processor 2, Die 1). The general tendency seems to be that the two cores from the remote die on the local processor (relative to the server core) are serviced first, then the core from the local die is serviced simultaneously (albeit faster) than the cores. Some subphases might appear when some cores serviced at a similar rate do not finish their execution at the exact same time.

Test-And-Test-And-Set (TATAS)

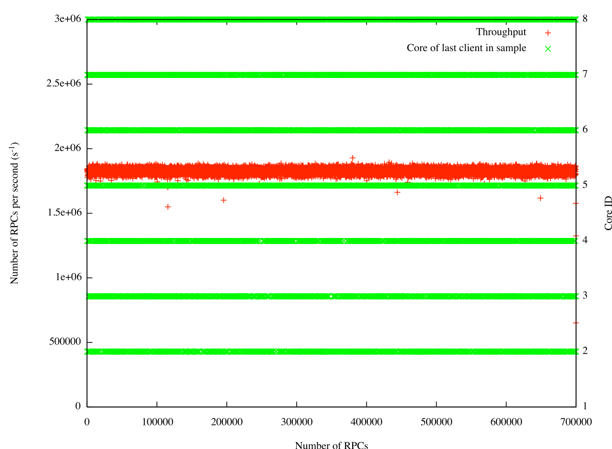
These experiments showed us that our first measurements were performed in a degenerate case in which fairness was not preserved. In order to solve this issue, our first approach was to use a Test-And-Test-And-Set (TATAS) loop instead of the Test-And-Set (TAS) loop we used until then (implemented with a Compare-And-Swap). In Figure 9, what we refer to a TAS loop is the loop from lines 7 to 10. Figure 13 shows the difference between a TAS loop and a TATAS loop. When using a TAS loop, all clients repeatedly try to set the value of the `global_sv` global variable to the address of the `local_sv` local variable in order to start a RPC. When using a TATAS loop, clients first check the value of the `global_sv` variables, and only



Delay = 1000 cycles (contended case)



Delay = 6100 cycles (maximum throughput)



Delay = 10000 cycles (high delay)

Figure 15: Sampled results for three different values of the waiting delay, using a TAS loop, on bossa. With a low delay, fairness is not ensured, which hints that there is a high level of contention. Increasing the delay improves fairness, until the benchmark is fully fair for the delay for which the throughput reaches its maximum (6100 cycles in our case, cf. Figure 14). Increasing the delay any further only lowers the throughput.

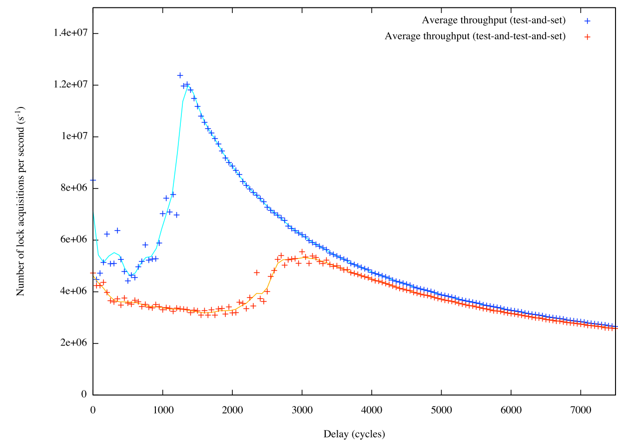


Figure 16: Number of lock acquisitions per second as a function of the delay, on bossa (7 clients).

try a CAS if this initial comparison hinted that the CAS would be successful. Using a TATAS is sometimes preferred to using a TAS, because it limits the number of CAS. A TATAS loops uses more comparisons, but comparisons are less costly than CASes, because they don't require to acquire writing rights to cache lines. However, since there is no way to ensure that the value of the synchronization variable will change between the comparison and the CAS, the actual results of using a TATAS loop are hard to predict, and depend largely on the underlying hardware.

Figure 12 shows the result of a sampled execution of our prototype modified to use a TATAS loop instead of a TAS loop. This technique greatly improves equity: all cores are able to acquire the RPC lock, even though local cores are privileged. However, the performance is worse, probably because cache lines are more contended with a TATAS loop.

Delays

Another approach we tried to limit contention over cache lines was to add a busy-wait loop after each RPC request on the client side in order to limit the contention on the cache lines. We performed experiments to find the optimal delay. The results are shown on Figure 14 (red/orange curve). There is a clear optimum, in both TAS and TATAS configurations, at 3000 and 1300 cycles respectively. With no delay or a small delay, the throughput is slow, and the variance is strong. It is also highly dependent on the initial conditions, which explains the issues we had with our first prototype.

Figure 15 shows sampled results of the execution of our prototype with a waiting delay of 1000, 6100 and 10000 cycles between each RPC request, using a TAS loop. With 1000 cycles, the experience is more fair to the clients than with no delay (cf. Figure 11, first graph): indeed, all clients get served from the first phase, albeit at varying rates (not all clients terminate at the same time). Increasing the delay improves fairness, until we reach the maximum throughput of 6100 cycles (cf. Figure 14). With 6100 cycles, the experience is perfectly fair to clients (the whole experiment is a single phase), and the throughput

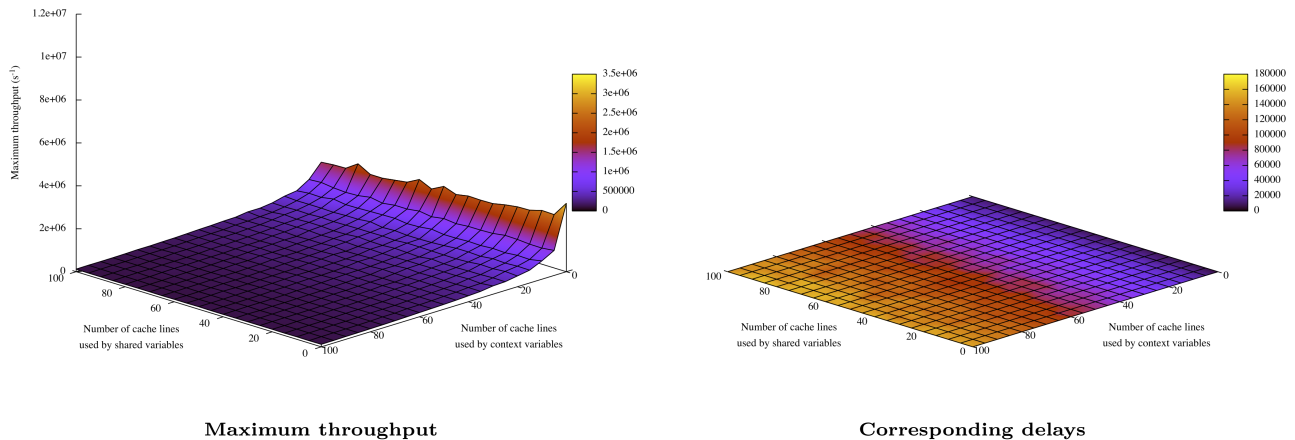


Figure 17: Maximum throughput and associated delays as a function of the number of context and shared variables, using RPCs (i.e. our implementation of Solution (b)) on bossa.

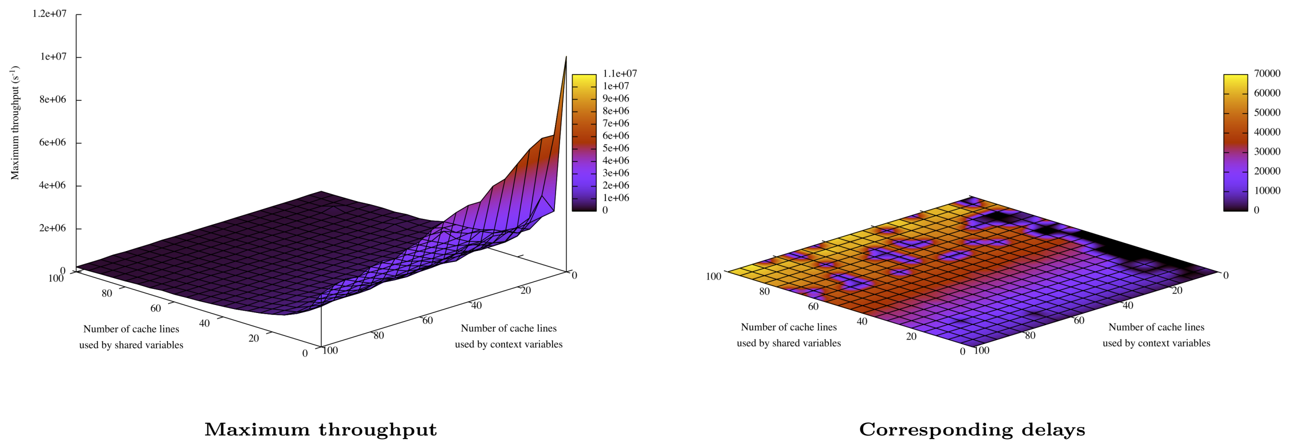


Figure 18: Maximum throughput and associated delays as a function of the number of context and shared variables, using locks (i.e. our implementation of Solution (a)) on bossa.

is maximal. When we increase the delay past this limit, fairness cannot be improved anymore and the throughput decreases. Our experiments show that the same observation can be made for the TATAS case: the maximum throughput corresponds to the minimal delay for which all clients are serviced fairly.

Waiting for a given delay on the client side is therefore a good way to ensure that our measurements are not limited by contention, and the maximum throughput of an experiment is a key value that characterizes the maximum number of RPCs per second that can be obtained in a given experimental case. This is a crucial point that will be used in later benchmarks (cf. section 3.4) to compare the results of such experiments.

3.3. Final benchmark

The final benchmark, whose code is presented in the appendix `benchmark.c`, implements two additional functionalities. It makes it possible (1) to execute critical sections on the clients based on lock acquisitions (this corresponds to Solution (a)) and (2) to access/modify context and shared variables from within the critical sections. This is the object of the two next subsections.

3.3.1. Executing critical sections locally

Up to now, we focused on the development of a benchmark that modelized our solution, i.e. “Solution (b)”. In order to compare Solution (a) and Solution (b), we needed to modelize Solution (a), i.e. the execution of critical sections on the clients themselves. We implemented this solution using a global lock that all clients try to acquire in order to execute their critical sections. After the execution of a critical section, the lock is released. The pseudo-code of the critical section for Solution (a) is shown in Figure 20.

Figure 16 shows the throughput of this solution with both a TAS and a TATAS loop. Once again, the TAS solution is more efficient than the TATAS solution. We obtain a maximum throughput of about 1.2e+07 critical sections per second as compared to 2.7+e06 critical sections per second when we used RPCs (cf. Figure 14). As

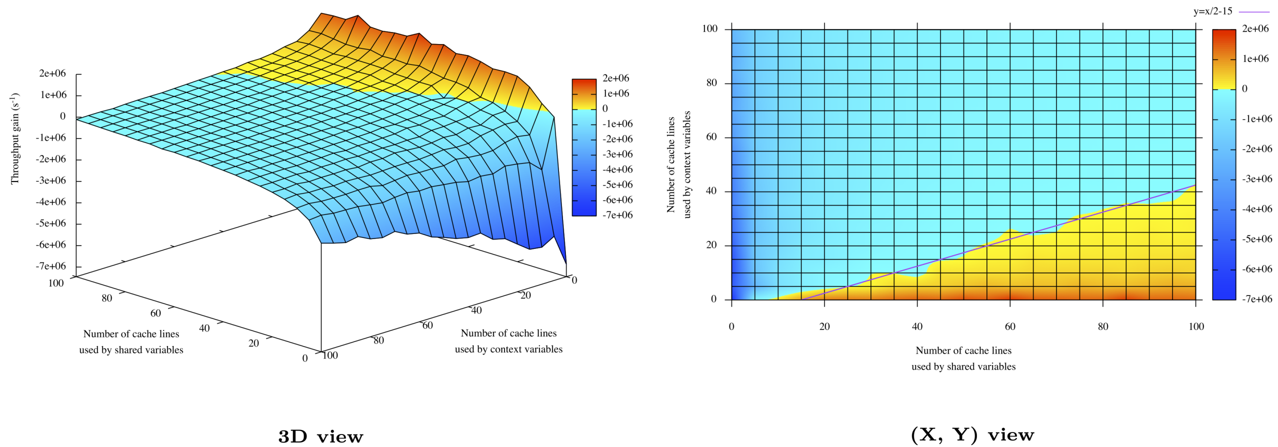


Figure 19: Maximum throughputs of our modelization of Solution (a) substracted from the maximum throughputs obtained for Solution (b), on bossa.

```
for (i = 0; i < number_of_rpcs; i++)
{
  *local_sv = 0;

  while (CAS(global_sv,
          0,
          1) == 0)
    pause;

  /* Protected code */

  global_sv = 1
}
```

Figure 20: Critical section for Solution (a).

explained before, we plan to compensate for this overhead with the potential performance improvement caused by the increase in locality when all critical sections are executed on a dedicated core. This is the object of the next subsection.

3.3.2. Context and shared variables

The last phase of the development of our benchmark was to add accesses to context and shared variables in and around the critical sections. We do not count the number of variables we access directly, since results could vary depending on where the variables are located in memory. Since the unit of data transfer between caches is the cache line, we allocate memory areas aligned with cache lines and access variables located every two cache lines (we skip one cache line between each variable to avoid issues with prefetching, as with `null_rpc.c`). One memory area per client is allocated for the context variables, and a global memory area is allocated for shared variables. Each memory access is a read/write access (++ operator from C). Here is the algorithm we chose for data accesses:

- Shared variables are accessed (i.e. incremented in a loop) once in each critical section, i.e. on the client side with our modelization of Solution (a) and on the server side with our modelization of Solution (b).
- Context variables are accessed once outside each critical section, and once in each critical section. We have to access the variables outside the critical sections to repatriate them from the cache hierarchy of the server to the cache hierarchy of the client before each critical section. This properly modelizes the behaviour context variables being accessed from whithin the critical section, and afterwards. This is different from the modelization that we used in Section 3.1 in which we considered context variables were ignored after having been used in a critical section. By considering that all context variables are always accessed after the critical sections, we modelized a worst case scenario, in which it is harder to compensate for the cost of the transfer of control and the transfer of context variables with the increased locality of shared variables.

Figure 17 shows the results of our benchmark for Solution (b). The first graph (labelled 'Maximum throughput') shows the value of the peak throughput for a given number of cache lines used by context and shared variables. The peak throughput is found as in the previous sections (cf. Figures 14 and 16), i.e. by finding the delay value for which the throughput is maximal. The graph shows that adding shared variables only has a slight impact on the throughput, which is what we expected: since all shared variables are in the local cache of the dedicated core, the overhead caused by shared variables is limited, thanks to the improved data locality. On the other hand, however, adding context variables is costly, because we have to fetch them from the client cores.

The second graph of Figure 17 show the corresponding delays we needed to reach the peak throughput. This is of no direct relevance on our final result, but allows us to better understand the inner workings of our benchmark and the validity of choosing a waiting delay as a way to avoid contention. The graph shows that the longer the experiment, the longer the delay needs to be. This is due to the fact that in a perfect scenario, all clients wait for the average time needed for the execution of a RPC: in this case, at the end of each RPC, a new unique client requests the

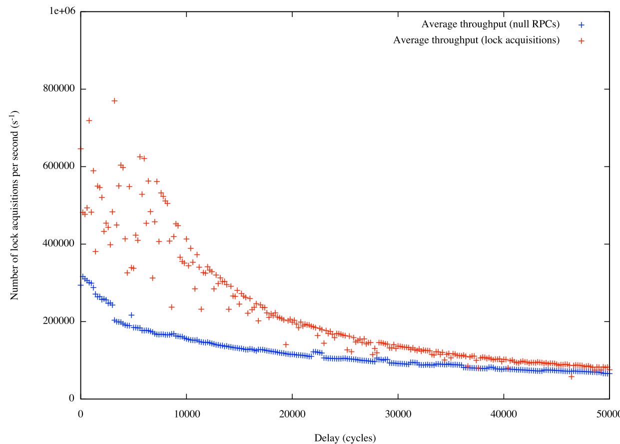


Figure 21: Throughput for Solutions (a) and (b) as a function of the delay, on amd48.

RPC, and there is no contention over the synchronization variables.

Similarly to Figure 17, Figure 18 shows the results of our benchmark for Solution (a). The peak throughput at (0,0) is much higher (10.0e+06 vs 3.1e+06), since in this case we do not migrate critical sections. The situation is the opposite with this solution compared to Solution (b): indeed, this time, adding context variable only induces limited overhead whereas the use of shared variables is prohibitive. This is of course the results we expected since the reason why we were not satisfied with Solution (a) was that the use of shared variable was too costly. The second graph of Figure 18 shows the delays corresponding to the maximum throughput and, unsurprisingly, they get higher when the critical sections get more costly.

In order to compare the two solutions, we substract the first curve from Figure 18 from the first curve from Figure 17. If the result is positive, this means that Solution (b) is more efficient than Solution (a). The result is shown on Figure 19 (using two different view). Our approach (Solution (b)) is more efficient than Solution (a) in the yellow/orange positive area.

This final result shows that, on bossa, with 7 clients on 8 cores, our approach (Solution (b)) is not profitable if less than 15 cache lines are shared. More precisely, it seems that Solution (b) is more efficient than Solution (a) when:

$$\boxed{n_{shared} > 2n_{context} + 15}$$

Where n_{shared} is the number of shared variables and $n_{context}$ is the number of context variables. This result is of course dependent on the hardware chosen and our experiment, but it gives us a general idea as to how efficient our approach can be.

3.4. Future experiments

The next phase will be to reproduce our experiments on amd48, in order to ensure that the results we obtain on a different architecture are similar. Moreover, since amd48 uses a large number of cores, it is a better example of a target architecture for our solution. Some experiments have already been reproduced: Figure 21 shows the throughput as a function of the delay for both Solution (a) and Solution (b) on amd48. For this experiment, we used the same settings as we did for bossa in figures 14 and 16, except there are now 47 threads instead of 7 (one per core, excluding the server). The variance is higher than in our experiments with bossa, probably due to the increased complexity of the hardware. Due to the increase in contention, the throughput is much lower (less than 1e+06 as compared to more than 1.2e+07 with Solution (b) on bossa). However, on both bossa and amd48, the ratio between the peak throughputs for Solution (a) and Solution (b) is still around 1/2, which tends to show that, even with the increased contention issues caused by a large number of cores, we are confident that the results of our benchmarks will be similar on amd48.

4. Conclusion

To ensure that our approach–outsourcing critical sections to dedicated cores–is viable, more experiments will be needed. First, we will have to reproduce our results on amd48, which has a different hardware architecture and a much higher number of cores. From the results on amd48 and bossa, we believe we will have a better idea of how viable our approach can be on various hardware configurations. Second, we will have to find a way to estimate the average number of cache lines used by shared and context variables in real-world Java programs. Finally, the parameter space in which our approach is profitable is too restricted, we will have to find automated ways to guess whether migrating critical sections is profitable before executing them.

For now, we remain optimistic, since the experiments we performed during this internship show that migrating critical sections to dedicated cores can be profitable under restricted conditions on the number of cache lines used by shared and context variables: for at least 15 cache lines used by context variables and a moderate number of context variables, outsourcing critical sections to a dedicated core is more efficient than executing them locally. These results show the potential of this approach which we will continue exploring in the context of a PhD thesis.

Bibliography

[1] PAPI, the Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. URPC: a toolkit for prototyping remote procedure calls. *The Computer Journal*, 39, no. 6:525–540, 1996.

[4] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12:271–307, 1994.

[5] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, 1994.

[6] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys conference*, pages 177–190, 2006.

[7] Y. Huang and C.V. Ravishankar. Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, 8, issue 1:37–55, 1990.

[8] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 341–354, 2006.

[9] Intel. Using Spin Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor.

[10] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. *ACM SIGPLAN Notices*, 45, issue 3:117–128, 2010.

[11] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, pages 125–138, 2010.

[12] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 221–234, 2009.

[13] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 2010 EuroSys conference*, pages 335–348, 2010.

[14] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwake Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 8:909–920, 1988.

[15] Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[16] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS ’05, pages 181–192, New York, NY, USA, 2005. ACM.

Appendix: source code

null_rpc.c

```c
/* ######################################################################### */
/* null_rpc.c                                                                */
/* (C) Jean-Pierre Lozi, 2010                                                */
/* ------------------------------------------------------------------------- */
/* This program should be compiled with the -O0 compiler flag.              */
/* ######################################################################### */
#define _GNU_SOURCE

#include <papi.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define PAUSE __asm__ __volatile__ ("rep; nop" : : );

#define PADDING 0
/* L2 cache lines are 32 bytes long, but they're fetched two at a time. */
/*#define PADDING 64*/

#define NUMBER_OF_ITERATIONS 1000000

void *client_main(void *ptr);
void *server_main(void *ptr);

int client_core, server_core, papi_cache_miss_level;

/* This structure contains the booleans we use as flags to signal events. */
struct flags {
    volatile unsigned char rpc_requested;
    volatile unsigned char padding[PADDING];
    volatile unsigned char done_with_rpc;
} flags;


int main(int argc, char **argv)
{
    if (argc != 4)
    {
        client_core = 0;
        server_core = 1;
        papi_cache_miss_level = 1;

        fprintf(stderr,
                "Usage : %s client_core server_core papi_cache_miss_level\n",
                argv[0]);
        fprintf(stderr,
                "Default values used (0, 1, 1)\n");
    }
    else
    {
        client_core = atoi(argv[1]);
        server_core = atoi(argv[2]);
        papi_cache_miss_level = atoi(argv[3]);

        if (papi_cache_miss_level < 1 || papi_cache_miss_level > 2) {
            fprintf(stderr,
                    "Usage : %s client_core server_core "
                    "papi_cache_miss_level\n",
                    argv[0]);
            fprintf(stderr,
                    "papi_cache_miss_level is invalid. Exiting.\n");
                exit(EXIT_FAILURE);
        }
    }

    /* We use both a client and a server. The client calls remote procedures
        from the server. */
    pthread_t client, server;

    flags.rpc_requested = 0;
    flags.done_with_rpc = 0;

    /* We create both threads. */
    if (pthread_create(&client, NULL, client_main, NULL) < 0)
        perror("pthread_create");

    if (pthread_create(&server, NULL, server_main, NULL) < 0)
        perror("pthread_create");

    /* We wait for both threads to finish their work. */
    if (pthread_join(client, NULL) < 0)
        perror("pthread_join");

    if (pthread_join(server, NULL) < 0)
        perror("pthread_join");

    /* Everything went as expected. */
    return EXIT_SUCCESS;
}

void *client_main(void *ptr)
{
    unsigned int j;
    int i;
    long long start_cycles, end_cycles;
    cpu_set_t cpuset;

    int event_set = PAPI_NULL;
    int events[1];
    long long values[1];

    /* We want to pin this thread to the right core. */
    CPU_ZERO(&cpuset);
    CPU_SET(client_core, &cpuset);
    /* We wish to avoid the server's core. */
    CPU_CLR(server_core, &cpuset);

    /* We set the thread's affinity. */
    if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset) < 0)
        perror("pthread_setaffinity_np");

    /* We monitor either L1 data cache misses or L2 data cache misses. */
    if (papi_cache_miss_level == 1)
        events[0] = PAPI_L1_STM;
    else
        events[0] = PAPI_L2_STM;

    /* We print miscellaneous informations about our computation. */
    printf("Client here. I'm running on core %d.\n", sched_getcpu());
    printf("Addresses : %x %x.\n", &flags.rpc_requested, &flags.done_with_rpc);
    printf("Starting %d RPCs.\n", NUMBER_OF_ITERATIONS);

    /* We initialize PAPI. */
    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
        perror("PAPI_library_init");

    if (PAPI_thread_init(pthread_self) != PAPI_OK)
        perror("PAPI_thread_init");

    /* We execute the first RPC separately, because it can be longer than others
       if the server isn't initialized yet.*/
    flags.rpc_requested = 1;

    while (!flags.done_with_rpc)
        PAUSE;

    flags.done_with_rpc = 0;

    /* We initialize PAPI and start recording events. */
    if (PAPI_create_eventset(&event_set) != PAPI_OK)
        perror("PAPI_create_eventset");

    if (PAPI_add_events(event_set, events, 1) != PAPI_OK)
        perror("PAPI_add_events");

    if (PAPI_start(event_set) != PAPI_OK)
        perror("PAPI_start");

    /* We start counting cycles. */
    start_cycles = PAPI_get_real_cyc();

    /* We execute the right number of RPCs. */
    for (i = 0 ; i < NUMBER_OF_ITERATIONS ; i++)
    {
        flags.rpc_requested = 1;

        while (!flags.done_with_rpc)
            PAUSE;

        flags.done_with_rpc = 0;

/*
        In order to measure events in a limited part of the loop, these two 'if'
        statements can be used around the instructions we wish to monitor.

        if (PAPI_reset(event_set) != PAPI_OK)
            perror("PAPI_reset");

        if (PAPI_accum(event_set, values) != PAPI_OK)
            perror("PAPI_accum");
*/
    }

    /* We stop the measurements. */
    end_cycles = PAPI_get_real_cyc();

    /* We read the counters. */
   if (PAPI_read(event_set, values) != PAPI_OK)
        perror("PAPI_read");

    printf("L%d cache misses : %lld\n", papi_cache_miss_level, values[0]);

    /* We print the result. */
    printf("Client : done. %lu cycles per null rpc. (total : %lld) %d\n",
           (end_cycles - start_cycles) / (long long)NUMBER_OF_ITERATIONS,
           end_cycles - start_cycles, j);

    /* We kill the server. */
    exit(EXIT_SUCCESS);

    return NULL;
}

void *server_main(void *ptr)
{
    unsigned int j;

    cpu_set_t cpuset;

    /* We want to pin this thread to the right core. */
```

```
201     CPU_ZERO(&cpuset);
202     CPU_SET(server_core, &cpuset);
203     /* We avoid the client's core. */
204     CPU_CLR(client_core, &cpuset);
205 
206     /* We set the thread's affinity. */
207     if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset) < 0)
208         perror("pthread_setaffinity_np");
209 
210     /* We print miscellaneous informations about our computation. */
211     printf("Server here. I'm running on core %d.\n", sched_getcpu());
212 
213     /* We start answering RPCs. */
214     for (;;)
215     {
216         while (!flags.rpc_requested)
217             PAUSE;
218 
219         flags.rpc_requested = 0;
220         flags.done_with_rpc = 1;
221     }
222 
223     return NULL;
224 }
```

benchmark.c

```c
/* ########################################################################## */
/* benchmark.c                                                                */
/* (C) Jean-Pierre Lozi, 2010                                                 */
/* -------------------------------------------------------------------------- */
/* This program should be compiled with the -O0 compiler flag.                */
/* ########################################################################## */
#define _GNU_SOURCE


/* ########################################################################## */
/* Headers                                                                    */
/* ########################################################################## */
#include <math.h>
#include <papi.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/resource.h>
#include <sys/time.h>


/* ########################################################################## */
/* Definitions                                                                */
/* ########################################################################## */
/*
    Use blocking locks?

    This parameter is experimental. When it's on, the benchmark uses blocking
    locks but only certain configurations are supported (no sampling, for
    instance). This mode cannot be accessed through command-line parameters
    because it is more of an experiment than anything else.
*/
/* #define USE_BLOCKING_LOCKS */

/* Default values for the command-line arguments. */
#define DEFAULT_NUMBER_OF_RUNS 1
#define DEFAULT_NUMBER_OF_ITERATIONS_PER_SAMPLE 10
#define DEFAULT_SERVER_CORE 0
#define DEFAULT_NUMBER_OF_ITERATIONS_PER_CLIENT 1000
#define DEFAULT_NUMBER_OF_CONTEXT_VARIABLES 0
#define DEFAULT_NUMBER_OF_SHARED_VARIABLES 0

/* Maximum number of cores, used to avoid malloc/realloc cycles. */
#define MAX_NUMBER_OF_CORES 1024

/* Maximum line size used when reading results from unix commands. */
#define MAX_LINE_SIZE 32

/* PAUSE instructions for spinlocks */
#define PAUSE __asm__ __volatile__ ("rep; nop" : : );

/* This macro performs the conversion between cycles and throughput. */
#define TO_THROUGHPUT(x) (cpu_frequency * 1000000 / (x))


/* ########################################################################## */
/* Global variables                                                           */
/* -------------------------------------------------------------------------- */
/* FIXME: some volatile keywords may not be necessary. We declare all global  */
/* variables as volatile for now.                                             */
/* ########################################################################## */
/* Environment data ========================================================= */
/* Number of cores */
volatile int number_of_cores;
/* CPU frequency, in MHz */
volatile float cpu_frequency;
/* This array maps physical to virtual core IDs. */
volatile int *physical_to_virtual_core_id;

/* Execution parameters ===================================================== */
/*
    Critical sections
*/
/* Critical sections can either be null RPCs (serviced by a single server) or
   lock acquisitions. */
typedef enum _critical_sections_type {
    NULL_RPCS,
    LOCK_ACQUISITIONS
} critical_sections_type_t;
volatile critical_sections_type_t critical_sections_type;

/*
    Execution mode
*/
/* Three execution modes are available :
    - MULTIPLE_RUNS_AVERAGED means that the test is run multiple times and the
      averaged results are returned (along with the variance and standard
      deviation).
    - SINGLE_RUN_SAMPLED means that the test is only run once, and statistics
      are gathered all along the execution.
    - SINGLE_RUN_ORDERED means that the test is only run once and that the order
      in which clients managed to execute their critical section is returned.
*/
typedef enum _execution_mode_t {
    MULTIPLE_RUNS_AVERAGED,
    SINGLE_RUN_SAMPLED,
    SINGLE_RUN_ORDERED
} execution_mode_t;
volatile execution_mode_t execution_mode;
/* Number of runs over which the results are averaged (used in the
   MULTIPLE_RUNS_AVERAGED mode only). */
volatile int number_of_runs;
/* Number of iterations per sample (used in the SINGLE_RUN_SAMPLED only). */
volatile int number_of_iterations_per_sample;

/*
    Execution settings
*/
/* Core on which the server runs */
volatile int server_core;
/* Number of clients */
volatile int number_of_clients;
/* Number of iterations per client */
volatile int number_of_iterations_per_client;
/* Delay between RPCs, in cycles. */
volatile int delay;
/* ROW mode activated? */
volatile int read_only_wait;
/* Number of context variables */
volatile int number_of_context_variables;
/* Number of shared variables */
volatile int number_of_shared_variables;

/*
    Measurements
*/
/* Should we count cycles or events? */
typedef enum _measurement_type {
    NUMBER_OF_CYCLES,
    NUMBER_OF_EVENTS
} measurement_type_t;
volatile measurement_type_t measurement_type;
/* ID of the monitored PAPI event if measurement_type = NUMBER_OF_EVENTS */
volatile int monitored_event_id;
/* Should we perform the measurements on the server or the clients? */
typedef enum _measurement_location {
    SERVER,
    CLIENTS
} measurement_location_t;
volatile measurement_location_t measurement_location;
/* Should we return the result in throughput or cycles? */
typedef enum _measurement_unit {
    THROUGHPUT,
    CYCLES_PER_ITERATION,
    TOTAL_CYCLES_MAX
} measurement_unit_t;
volatile measurement_unit_t measurement_unit;

/* Execution variables ====================================================== */
/* Adresses of the rpc_done addresses for each thread */
volatile void ** volatile rpc_done_addresses;
/* Synchronization variables for the barrier between the server and the
   clients, before the main loop */
volatile int * volatile ready;

/* Data specific to the MULTIPLE_RUNS_AVERAGED mode */
/* Results (cycles or number of events) for each iteration */
volatile double * volatile iteration_result;

/* Data specific to the SINGLE_RUN_SAMPLED mode */
/* Number of iterations per sample */
volatile float * volatile multiple_samples_results;
/* Address of the rpc_done variable from the core whose RPC was serviced last.
   Translated into the core ID at the end of the computation. */
volatile void ** volatile multiple_samples_rpc_done_addrs;

/* Data specific to the SINGLE_RUN_ORDERED mode */
/* Order in which the RPCs are processed */
volatile int * volatile order;

/* Global crical section variables ========================================== */
/* This structure models a cache line. It should be allocated on a 64-byte
   boundary. */
typedef char cache_line_t[64];

/*
   This structure models the data passed to threads upon their creation.
   Variables for both the 'NULL RPCs' and 'lock acquisitions' modes are passed
   in order to simplify the code.
*/
typedef struct _thread_arguments_block_t {
    /* Logical thread id (local, not provided by pthreads). */
    int id;

    /* 'NULL RPCs' mode synchronization variables =========================== */
    /*
        Global synchronization variable used to limit the number of NULL RPCs
        serviced concurrently to 1. Its value can either be:
            - 0 if no RPC is requested.
            - The address to a client-bound variable that shall be set to 1 upon
              completion of the RPC otherwise.
    */
    volatile uint64_t *volatile *null_rpc_global_sv;
    /*
        Local synchronization variable set to a >= 0 value by the server to
        signal the completion of a RPC.
    */
    volatile uint64_t *null_rpc_local_sv;

    /* 'Lock acquisitions' mode synchronization variables =================== */
    /*
        Global lock repeatedly acquired by all clients.
    */
    volatile uint64_t *lock_acquisitions_global_sv;

    volatile uint64_t *context_variables_global_memory_area;
    volatile uint64_t *shared_variables_memory_area;
} thread_arguments_block_t;

/* Mutexes and conditions used by non-blocking locks ======================== */
#ifdef USE_BLOCKING_LOCKS
pthread_mutex_t mutex_rpc_done_addr_not_null;
pthread_cond_t cond_rpc_done_addr_not_null;
pthread_mutex_t mutex_rpc_done_positive;
pthread_cond_t cond_rpc_done_positive;
pthread_mutex_t mutex_rpc_done_addr_null;
pthread_cond_t cond_rpc_done_addr_null;
#endif
```

```c

/* ########################################################################### */
/* Prototypes                                                                  */
/* ########################################################################### */
void *client_main(void *thread_arguments_block);
void *server_main(void *thread_arguments_block);

void *alloc(int size);
void get_cpu_info();

void error(char *reason);
void wrong_parameters_error(char *application_name);

/* ########################################################################### */
/* Functions                                                                   */
/* ########################################################################### */
/* Main function */
int main(int argc, char **argv)
{
    int i, j;
    int command;
    int result;
    double *results;
    double *average, *variance;
    int end_output_with_a_newline = 1;
    int compute_standard_deviation_and_variance = 0;
    int number_of_samples = 0;

    /* We use both a server and number_of_clients clients. The clients call
       remote procedures from the server. */
    pthread_t server;
    pthread_t *clients;

    /* This array contains the argument blocks passed to the clients. */
    thread_arguments_block_t *svs_memory_area,
                             *thread_argument_blocks,
                             *context_variables_global_memory_area,
                             *shared_variables_memory_area;

    /* We get the clock speed and the core ordering. */
    get_cpu_info();

    /*
       Command-line arguments

       Critical sections
       =================

       -R
       Perform null RPCs. This is the default behavior.

       -L
       Perform lock acquisitions instead of null RPCs. In this mode, there is
       no server.

       If both -R and -L are specified, the last one wins.

       Execution mode
       ==============

       -A number_of_runs
       Return the average number of cycles, with variance and standard
       deviation, over the given number of runs. This is the default mode.

       -S number_of_iterations_per_sample
       Return sampled results over a single run, using the given number of
       iterations per sample.

       -O
       In this mode, each client enters its critical section only once, and the
       order in which the clients were serviced is returned. Only one run is
       performed and the results are not averaged.

       If several execution modes are specified, the last one wins.

       Execution settings
       ==================

       -s core
       Core on which the server thread runs if the critical section consists of
       performing null RPCs, or core of the first client if the critical section
       consists of lock acquisitions.

       -c number_of_clients
       Number of clients.

       -n number_of_iterations_per_client
       Number of RPCs/lock acquisitions per client.

       -d delay
       Number of cycles wasted (busy waiting) by clients after a RPC is
       serviced. If 0, no cycles are wasted (apart from the execution time of a
       comparison), otherwise the accuracy of the delay is within 50~150 cycles.

       -r
       Read-only wait mode : wait using comparisons, and only perform a CAS
       after a successful comparison (test-and-test-and-set).

       -C number_of_context_variables

       Number of context (local) variables.

       -S number_of_shared_variables

       Number of shared (global) variables.

       Measurements
       ============

       -e event_type
       Count events of type event_type instead of cycles. Event_type is the type
       of a PAPI event.

       -m
       Count cycles (or events) on the clients instead of the server. This is
       always the case if -L is on.

       -y
       Returns results in cycles per iterations instead of the number of
       iterations per second.

       -t
       Returns results in total cycles instead of the number of iterations per
       second. Only the maximum value is returned.

       Output
       ======

       -v
       Return the standard deviation and variance if applicable.

       -i
       End the results with a newline. This is left as an option, to allow for
       more flexibility with the CSV results.
    */

    opterr = 0;

    /* Default values */
    critical_sections_type = NULL_RPCS;

    execution_mode = MULTIPLE_RUNS_AVERAGED;
    number_of_runs = DEFAULT_NUMBER_OF_RUNS;
    number_of_iterations_per_sample = DEFAULT_NUMBER_OF_ITERATIONS_PER_SAMPLE;

    server_core = DEFAULT_SERVER_CORE;
    number_of_clients = - 1;
    number_of_iterations_per_client = DEFAULT_NUMBER_OF_ITERATIONS_PER_CLIENT;
    delay = 0;
    read_only_wait = 0;

    number_of_context_variables = DEFAULT_NUMBER_OF_CONTEXT_VARIABLES;
    number_of_shared_variables = DEFAULT_NUMBER_OF_SHARED_VARIABLES;

    measurement_type = NUMBER_OF_CYCLES;
    measurement_location = SERVER;
    measurement_unit = THROUGHPUT;

    while ((command = getopt(argc, argv, "RLA:S:Os:c:n:d:rl:g:tTe:myvi")) != -1)
    {
        switch (command)
        {
            /*
                Critical sections
            */
            case 'R':
                /* Actually useless (default value). */
                critical_sections_type = NULL_RPCS;
                break;
            case 'L' :
                critical_sections_type = LOCK_ACQUISITIONS;
                break;

            /*
                Execution mode
            */
            case 'A':
                execution_mode = MULTIPLE_RUNS_AVERAGED;
                number_of_runs = atoi(optarg);
                break;

            case 'S':
                execution_mode = SINGLE_RUN_SAMPLED;
                number_of_iterations_per_sample = atoi(optarg);
                break;

            case 'O':
                execution_mode = SINGLE_RUN_ORDERED;
                break;

            /*
                Execution settings
            */
            case 's':
                server_core = atoi(optarg);
                break;

            case 'c':
                number_of_clients = atoi(optarg);
                break;

            case 'n':
                number_of_iterations_per_client = atoi(optarg);
                break;

            case 'd':
                delay = atoi(optarg);
                break;

            case 'r':
                read_only_wait = 1;
                break;

            case 'l':
                number_of_context_variables = atoi(optarg);
                break;

            case 'g':
                number_of_shared_variables = atoi(optarg);
                break;

            /*
                Measurements
            */
            case 'e':
                measurement_type = NUMBER_OF_EVENTS;
                monitored_event_id = strtol(optarg, NULL, 16);
                break;

            case 'm':
```

```
453                 measurement_location = CLIENTS;
454                 break;
455 
456             case 'y':
457                 measurement_unit = CYCLES_PER_ITERATION;
458                 break;
459 
460             case 't':
461                 measurement_unit = TOTAL_CYCLES_MAX;
462                 break;
463 
464             /*
465                 Output
466             */
467             case 'v':
468                 compute_standard_deviation_and_variance = 1;
469                 break;
470 
471             case 'i':
472                 end_output_with_a_newline = 0;
473                 break;
474 
475             /*
476                 Other
477             */
478             case '?': wrong_parameters_error(argv[0]);
479             default : error("getopt");
480         }
481     }
482 
483     /* Default values for the number of clients */
484     if (number_of_clients <= 0)
485     {
486         number_of_clients = number_of_cores;
487 
488         if (critical_sections_type == NULL_RPCS)
489         {
490             number_of_clients--;
491         }
492     }
493 
494     if (critical_sections_type == LOCK_ACQUISITIONS)
495     {
496         /* Lock acquisitions are only compatible with the MULTIPLE_RUNS_AVERAGED
497            execution mode. */
498         if (execution_mode != MULTIPLE_RUNS_AVERAGED)
499             error("configuration not supported");
500 
501         /* In this mode, measurements take place on the clients. */
502         measurement_location = CLIENTS;
503     }
504 
505     if (execution_mode == SINGLE_RUN_SAMPLED)
506     {
507         /* For now, the SINGLE_RUN_SAMPLED execution mode in incompatible with
508            blocking locks. */
509 #ifdef USE_BLOCKING_LOCKS
510         error("configuration not supported");
511 #endif
512 
513         /* For now, in the SINGLE_RUN_SAMPLED mode, events can't be monitored.
514            Also, all measurements must take place on the server. */
515         if (measurement_type != NUMBER_OF_CYCLES
516             && measurement_location != SERVER)
517             error("configuration not supported");
518 
519         /* Single run */
520         number_of_runs = 1;
521 
522         if (number_of_iterations_per_sample >
523             number_of_iterations_per_client * number_of_clients)
524             error("too many iterations per sample");
525     }
526 
527     /* In the SINGLE_RUN_ORDERED execution mode... */
528     if (execution_mode == SINGLE_RUN_ORDERED)
529     {
530         /* ...there's just one iteration per client... */
531         number_of_iterations_per_client = 1;
532 
533         /* ...and a single run. */
534         number_of_runs = 1;
535     }
536 
537     /* We need the maximum priority (for performance measurements). */
538     if (setpriority(PRIO_PROCESS, 0, -20) < 0)
539         error("setpriority");
540 
541     /* ================================================================== */
542     /* [v] Memory allocations                                             */
543     /* ================================================================== */
544     /* The following arrays' sizes depend on the parameters, we need to allocate
545        them dynamically. */
546     clients                = alloc(number_of_clients * sizeof(pthread_t));
547     thread_argument_blocks = alloc((number_of_clients + 1) *
548                                    sizeof(thread_arguments_block_t));
549     ready                  = alloc((number_of_clients + 1) * sizeof(int));
550     iteration_result       = alloc((number_of_clients + 1) * sizeof(double));
551     results                = alloc((number_of_clients + 1) *
552                                  (number_of_runs * sizeof(double)));
553     average                = alloc((number_of_clients + 1) * sizeof(double));
554     variance               = alloc((number_of_clients + 1) * sizeof(double));
555 
556     /* If we need to return the number in which RPCs are serviced... */
557     if (execution_mode == SINGLE_RUN_ORDERED)
558     {
559         /* ...we allocate the 'order' array dynamically. */
560         order = alloc(number_of_clients * sizeof(double));
561     }
562 
563     /* We always allocate the sampling-related arrays. */
564     rpc_done_addresses = alloc((number_of_clients + 1) * sizeof(void *));
565 
566     number_of_samples = number_of_iterations_per_client * number_of_clients
567                         / number_of_iterations_per_sample;
568 
569     multiple_samples_results = alloc(number_of_samples * sizeof(double));
570 
571     multiple_samples_rpc_done_addrs =
572         alloc(number_of_samples * sizeof(void *));
573 
574     /* Memory area containing the synchronization variables. Each synchronisa-
575        tion variable is allocated on its own pair of cache lines. */
576     result = posix_memalign((void **)&svs_memory_area,
577                             128,
578                             (number_of_clients + 1) * 2 * sizeof(cache_line_t));
579 
580     if (result < 0 || svs_memory_area == NULL)
581         error("memalign");
582 
583     result = posix_memalign((void **)&context_variables_global_memory_area,
584                             128,
585                             (number_of_clients + 1) *
586                             (number_of_context_variables + 1) *
587                             2 * sizeof(cache_line_t));
588 
589     if (result < 0 || svs_memory_area == NULL)
590         error("memalign");
591 
592     result = posix_memalign((void **)&shared_variables_memory_area,
593                             128,
594                             (number_of_shared_variables) *
595                             2 * sizeof(cache_line_t));
596 
597     if (result < 0 || svs_memory_area == NULL)
598         error("memalign");
599 
600     /* ================================================================== */
601     /* [^] Memory allocations                                             */
602     /* ================================================================== */
603 
604     /* We initialize PAPI. */
605     if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
606         error("PAPI_library_init");
607 
608     /* If we use blocking locks, then some mutexes and conditions should be
609        initialized. */
610 #ifdef USE_BLOCKING_LOCKS
611     pthread_mutex_init(&mutex_rpc_done_addr_not_null, NULL);
612     pthread_mutex_init(&mutex_rpc_done_positive, NULL);
613     pthread_mutex_init(&mutex_rpc_done_addr_null, NULL);
614 
615     if (pthread_cond_init(&cond_rpc_done_addr_not_null, NULL) < 0
616         || pthread_cond_init(&cond_rpc_done_positive, NULL) < 0
617         || pthread_cond_init(&cond_rpc_done_addr_null, NULL) < 0)
618         error("pthread_cond_init");
619 #endif
620 
621     /* We iterate to average the results. */
622     for (i = 0; i < number_of_runs; i++)
623     {
624         /* No rpc started for now. */
625         *((uint64_t **)svs_memory_area) = 0;
626 
627         /* None of the clients are ready. */
628         for (j = 0; j <= number_of_clients; j++)
629             ready[j] = 0;
630 
631         /* Initialisations related to sampling. */
632         if (execution_mode == SINGLE_RUN_SAMPLED)
633         {
634             for (j = 0; j <= number_of_samples; j++)
635                 multiple_samples_results[j] = 0;
636 
637             for (j = 0; j <= number_of_samples; j++)
638                 multiple_samples_rpc_done_addrs[j] = 0;
639         }
640 
641         /* If we perform null RPCs in the critical sections... */
642         if (critical_sections_type == NULL_RPCS)
643         {
644             thread_argument_blocks[0].id = 0;
645             thread_argument_blocks[0].null_rpc_global_sv =
646                 (volatile uint64_t * volatile *)(&svs_memory_area[0]);
647             thread_argument_blocks[0].null_rpc_local_sv = NULL;
648 
649             /* FIXME: global variable? */
650             thread_argument_blocks[0].context_variables_global_memory_area =
651                 (volatile uint64_t *)(context_variables_global_memory_area);
652             thread_argument_blocks[0].shared_variables_memory_area =
653                 (volatile uint64_t *)(shared_variables_memory_area);
654 
655             /* ...we create a server thread. */
656             if (pthread_create(&server, NULL, server_main,
657                                &thread_argument_blocks[0]) < 0)
658                 error("pthread_create");
659         }
660 
661         /* We create the client threads. */
662         for (j = 0; j < number_of_clients; j++)
663         {
664             thread_argument_blocks[j + 1].id = j;
665             thread_argument_blocks[j + 1].null_rpc_global_sv =
666                 (volatile uint64_t * volatile *)(&svs_memory_area[0]);
667             thread_argument_blocks[j + 1].null_rpc_local_sv =
668                 (volatile uint64_t *)
669                 ((uint64_t)context_variables_global_memory_area +
670                            j * (number_of_context_variables + 1) *
671                            2 * sizeof(cache_line_t));
672 
673             thread_argument_blocks[j + 1].lock_acquisitions_global_sv =
674                 (volatile uint64_t *)(&svs_memory_area[0]);
675 
676             thread_argument_blocks[j + 1].context_variables_global_memory_area =
677                 (volatile uint64_t *)
678                 ((uint64_t)context_variables_global_memory_area);
679             thread_argument_blocks[j + 1].shared_variables_memory_area =
680                 (volatile uint64_t *)(shared_variables_memory_area);
681 
682             /* We send i directly instead of sending a pointer to avoid having
```

```
            to create a variable per thread containing the value. */
        if (pthread_create(&clients[j], NULL, client_main,
                           &thread_argument_blocks[j + 1]) < 0)
            error("pthread_create");
    }

    /* If there is a server thread... */
    if (critical_sections_type == NULL_RPCS)
    {
        /* ...we wait for it to finish its work. */
        if (pthread_join(server, NULL) < 0)
            error("pthread_join");
    }

    /* We wait for the clients to finish their work. */
    for (j = 0; j < number_of_clients; j++)
    {
        if (pthread_join(clients[j], NULL) < 0)
            error("pthread_join");
    }

    /* We save the results. */
    for (j = 0 ; j <= number_of_clients; j++)
    {
        if (measurement_unit == THROUGHPUT)
            iteration_result[j] = TO_THROUGHPUT(iteration_result[j]);

        results[j * number_of_runs + i] = iteration_result[j];
    }
}

/* In the ordered mode, we print the order in which the RPCs were
   serviced. */
if (execution_mode == SINGLE_RUN_ORDERED)
{
    for (i = 0; i < number_of_clients; i++)
    {
        j = 0;

        while (order[j] != i)
            j++;

        printf("%d", j);
    }

    printf(",");
}

/* If sampling is on, we print the result of each sample followed by the ID
   of the last core whose RPC has been serviced. */
if (execution_mode == SINGLE_RUN_SAMPLED)
{
    for (i = 0; i < number_of_samples; i++)
    {
        for (j = 0; j < number_of_clients + 1; j++)
        {
            if (multiple_samples_rpc_done_addrs[i] == rpc_done_addresses[j])
                break;
        }

        /* We convert the result to the right unit if needed. */
        if (measurement_unit == THROUGHPUT)
            multiple_samples_results[i] =
                TO_THROUGHPUT(multiple_samples_results[i]);

        /* We return the result. */
        printf("%f,%d,\n", multiple_samples_results[i], j);
    }
}

if (measurement_unit==TOTAL_CYCLES_MAX)
{

    for (i = 0; i < number_of_runs; i++)
    {
        double max = 0;

        for (j = 1; j <= number_of_clients; j++)
        {
            if (max < results[j * number_of_runs + i])
                max = results[j * number_of_runs + i];
        }

        results[i] = TO_THROUGHPUT(max / (number_of_iterations_per_client *
                                          number_of_clients));
    }
}

/* We compute the average and the variance (if needed). */
for (j = 0; j <= number_of_clients; j++)
{
    /* Initialization */
    for (i = 0; i < number_of_runs; i++)
    {
        average[j] = 0;
        variance[j] = 0;
    }

    /* We compute the average. */
    for (i = 0; i < number_of_runs; i++)
    {
        average[j] += results[j * number_of_runs + i];
    }

    average[j] /= number_of_runs;

    /* We compute the variance (only if needed). */
    if (compute_standard_deviation_and_variance)
    {
        for (i = 0; i < number_of_runs; i++)
        {
            variance[j] +=
                (results[j * number_of_runs + i] - average[j]) *
                (results[j * number_of_runs + i] - average[j]);
        }

        variance[j] /= number_of_runs;
    }
}

/* We print the result. If the measurements took place on the server... */
if (measurement_location == SERVER || measurement_unit == TOTAL_CYCLES_MAX)
{
    /* ...we return the average... */
    printf("%f,", average[0]);

    if (compute_standard_deviation_and_variance)
        /* ...and, if needed, the variance and standard deviation. */
        printf("%f,%f,", variance[0], sqrt(variance[0]));
}
/* Otherwise... */
else
{
    /* ...it's the same, but for each client. */
    for (j = 1; j <= number_of_clients; j++)
    {
        printf("%f,", average[j]);

        if (compute_standard_deviation_and_variance)
            /* ...and, if needed, the variance and standard deviation. */
            printf("%f,%f,", variance[j], sqrt(variance[j]));
    }
}

/* If sampling is enabled, we need to specify a value for the core ID of the
   average. We choose -1. */
if (execution_mode == SINGLE_RUN_SAMPLED)
    printf("%d,", -1);

/* Should we end the input with a newline? */
if (end_output_with_a_newline)
    printf("\n");
else
    /* If we don't, we make sure to flush the standard output. */
    fflush(NULL);

/* ================================================================== */
/* [v] Cleanup (unnecessary)                                          */
/* ================================================================== */
free((int *)physical_to_virtual_core_id);
free(clients);
free(thread_argument_blocks);
free((int *)ready);
free((double *)iteration_result);
free(results);
free(average);
free(variance);
if (execution_mode == SINGLE_RUN_ORDERED) free((int *)order);
if (execution_mode == SINGLE_RUN_SAMPLED)
{
    free(rpc_done_addresses);
    free((float *)multiple_samples_results);
    free((void **)multiple_samples_rpc_done_addrs);
}

#ifdef USE_BLOCKING_LOCKS
    if (pthread_mutex_destroy(&mutex_rpc_done_addr_not_null) < 0
     || pthread_mutex_destroy(&mutex_rpc_done_positive) < 0
     || pthread_mutex_destroy(&mutex_rpc_done_addr_null) < 0)
     ERROR("pthread_mutex_destroy");

     if (pthread_cond_destroy(&cond_rpc_done_addr_not_null) < 0
      || pthread_cond_destroy(&cond_rpc_done_positive) < 0
      || pthread_cond_destroy(&cond_rpc_done_addr_null) < 0)
      ERROR("pthread_cond_destroy");
#endif
    /* ================================================================== */
    /* [^] Cleanup (unnecessary)                                          */
    /* ================================================================== */

    /* Everything went as expected. */
    return EXIT_SUCCESS;
}

/* Function executed by the clients */
void *client_main(void *thread_arguments_block_pointer)
{
    int i, k;

    /* This is needed because the CAS function requires an address. */
    const int one = 1;

    /* We avoid using global variables in (and around) the critical section
       to limit data cache misses. */
    const int local_number_of_iterations_per_client =
        number_of_iterations_per_client;
    const int local_delay = delay;

    const int local_number_of_context_variables = number_of_context_variables;
    const int local_number_of_shared_variables = number_of_shared_variables;

    /* Argument block */
    thread_arguments_block_t thread_arguments_block;

    /* We copy the contents of the argument block into these variables to
       improve the readability of the code. */
    volatile uint64_t *volatile *null_rpc_global_sv;
    volatile uint64_t *null_rpc_local_sv;
    volatile uint64_t *lock_acquisitions_global_sv;

    volatile uint64_t *context_variables_local_memory_area,
                      *shared_variables_memory_area;

    cpu_set_t cpuset;
    int client_id, client_core;

    /* PAPI variables */
    int event_set = PAPI_NULL;
    int events[1];
    long long values[1];
```

```
913     long long start_cycles = 0, end_cycles;
914     long long cycles;
915
916     /* We get the client id. */
917     thread_arguments_block =
918         *((thread_arguments_block_t *)thread_arguments_block_pointer);
919
920     client_id = thread_arguments_block.id;
921     null_rpc_local_sv = thread_arguments_block.null_rpc_local_sv;
922     null_rpc_global_sv = thread_arguments_block.null_rpc_global_sv;
923     lock_acquisitions_global_sv =
924         thread_arguments_block.lock_acquisitions_global_sv;
925
926     context_variables_local_memory_area =
927         (volatile uint64_t *)
928         ((uint64_t)thread_arguments_block.context_variables_global_memory_area +
929         2 * sizeof(cache_line_t) * (local_number_of_context_variables + 1) *
930         client_id + 2 * sizeof(cache_line_t));
931     shared_variables_memory_area =
932         thread_arguments_block.shared_variables_memory_area;
933
934     /* We compute a proper core for the client to execute on. */
935     if (critical_sections_type == NULL_RPCS)
936     {
937         client_core = (server_core + 1 + (client_id % (number_of_cores - 1))) %
938                       number_of_cores;
939     }
940     else
941     {
942         client_core = (server_core + client_id) % number_of_cores;
943     }
944
945     /* We pin this thread to the right core. */
946     CPU_ZERO(&cpuset);
947     CPU_SET(physical_to_virtual_core_id[client_core], &cpuset);
948
949     /* We set the thread's affinity. */
950     if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset) < 0)
951         error("pthread_setaffinity_np");
952
953     /* If sampling is on... */
954     if (execution_mode == SINGLE_RUN_SAMPLED)
955     {
956         /* ...we register the address of our rpc_done local variable. */
957         rpc_done_addresses[client_core] = null_rpc_local_sv;
958     }
959
960     if (measurement_location == CLIENTS)
961     {
962         /* We will use PAPI in this thread. */
963         PAPI_thread_init((unsigned long (*)(void))pthread_self);
964
965         /* Additional initialization is needed if we are counting events. */
966         if (measurement_type == NUMBER_OF_EVENTS)
967         {
968             events[0] = monitored_event_id;
969
970             if (PAPI_create_eventset(&event_set) != PAPI_OK)
971                 error("PAPI_create_eventset");
972             if (PAPI_add_events(event_set, events, 1) != PAPI_OK)
973                 error("PAPI_add_events");
974
975             /* This seemingly helps increasing PAPI's accuracy. */
976             if (PAPI_start(event_set) != PAPI_OK)
977                 error("PAPI_start");
978             if (PAPI_stop(event_set, values) != PAPI_OK)
979                 error("PAPI_stop");
980         }
981     }
982
983     /* We're ready. */
984     ready[client_id + 1] = 1;
985
986     /* Synchronization barriers */
987     if (critical_sections_type == LOCK_ACQUISITIONS)
988     {
989         for (i = 1; i <= number_of_clients; i++)
990             while (!ready[i])
991                 PAUSE;
992     }
993     if (critical_sections_type == NULL_RPCS && measurement_location == CLIENTS)
994     {
995         for (i = 0; i <= number_of_clients; i++)
996             while (!ready[i])
997                 PAUSE;
998     }
999
1000    if (measurement_location == CLIENTS)
1001    {
1002        /* Does the user wish to measure the number of elapsed cycles? */
1003        if (measurement_type == NUMBER_OF_CYCLES)
1004        {
1005            /* If so, get the current cycle count. */
1006            start_cycles = PAPI_get_real_cyc();
1007        }
1008        else
1009        {
1010            /* Otherwise we start counting events. */
1011            if (PAPI_start(event_set) != PAPI_OK)
1012                error("PAPI_start");
1013        }
1014    }
1015
1016 /* ######################################################################## */
1017 /* # Main loop (client)                                                   # */
1018 /* ######################################################################## */
1019 /* First implementation : using spinlocks. */
1020 #ifndef USE_BLOCKING_LOCKS
1021     /* NULL RPCs */
1022     if (critical_sections_type == NULL_RPCS)
1023     {
1024         /* We execute number_of_iterations_per_client RPCs. */
1025         for (i = 0; i < local_number_of_iterations_per_client; i++)
1026         {
1027             *null_rpc_local_sv = -1;
1028
1029             /* We access context variables */
1030             for (k = 0; k < local_number_of_context_variables; k++)
1031                 (*((volatile uint64_t *)
1032                   ((uint64_t)context_variables_local_memory_area +
1033                   k * 2 * sizeof(cache_line_t))))++;
1034
1035             if (!read_only_wait)
1036             {
1037                 /* We use a compare and swap to avoid mutexes. */
1038                 while (!__sync_bool_compare_and_swap(null_rpc_global_sv,
1039                                                      0,
1040                                                      null_rpc_local_sv))
1041                     PAUSE;
1042             }
1043             else
1044             {
1045                 for (;;)
1046                 {
1047                     if (!(*null_rpc_global_sv))
1048                     {
1049                         if (!__sync_bool_compare_and_swap(null_rpc_global_sv,
1050                                                           0,
1051                                                           null_rpc_local_sv))
1052                             continue;
1053                         else break;
1054                     }
1055
1056                     PAUSE;
1057                 }
1058             }
1059
1060             while (*null_rpc_local_sv < 0)
1061                 PAUSE;
1062
1063             /* We could avoid the delay introduced by the test by moving the
1064                if outside of the loop, however, tests show that no delay or
1065                a small delay doesn't alter the results significantly. */
1066             if (local_delay > 0)
1067             {
1068                 /* Delay */
1069                 cycles = PAPI_get_real_cyc();
1070                 while ((PAPI_get_real_cyc() - cycles) < local_delay)
1071                     ;
1072             }
1073         }
1074     }
1075     /* Lock acquisitions */
1076     else
1077     {
1078         if (execution_mode != SINGLE_RUN_SAMPLED)
1079         {
1080             for (i = 0; i < local_number_of_iterations_per_client; i++)
1081             {
1082                 /* We access context variables */
1083                 for (k = 0; k < local_number_of_context_variables; k++)
1084                     (*((volatile uint64_t *)
1085                       ((uint64_t)context_variables_local_memory_area +
1086                       k * 2 * sizeof(cache_line_t))))++;
1087
1088                 if (!read_only_wait)
1089                 {
1090                     while (!__sync_bool_compare_and_swap
1091                             (lock_acquisitions_global_sv, 0, &one))
1092                         PAUSE;
1093                 }
1094                 else
1095                 {
1096                     for (;;)
1097                     {
1098                         if (!*null_rpc_global_sv)
1099                         {
1100                             if (!__sync_bool_compare_and_swap
1101                                     (lock_acquisitions_global_sv, 0, &one))
1102                                 continue;
1103
1104                             else break;
1105                         }
1106
1107                         PAUSE;
1108                     }
1109                 }
1110
1111                 /* We access context variables */
1112                 for (k = 0; k < local_number_of_context_variables; k++)
1113                     (*((volatile uint64_t *)
1114                       ((uint64_t)context_variables_local_memory_area +
1115                       k * 2 * sizeof(cache_line_t))))++;
1116
1117                 /* We access shared variables */
1118                 for (k = 0; k < local_number_of_shared_variables; k++)
1119                     (*((volatile uint64_t *)
1120                       ((uint64_t)shared_variables_memory_area +
1121                       k * 2 * sizeof(cache_line_t))))++;
1122
1123                 *lock_acquisitions_global_sv = 0;
1124
1125                 if (local_delay > 0)
1126                 {
1127                     /* Delay */
1128                     cycles = PAPI_get_real_cyc();
1129                     while ((PAPI_get_real_cyc() - cycles) < local_delay)
1130                         ;
1131                 }
1132             }
1133         }
1134     }
1135 /* Second implementation : using blocking locks. */
1136 #else
1137     /* We execute number_of_iterations_per_client RPCs. */
1138     for (i = 0; i < local_number_of_iterations_per_client; i++)
1139     {
1140         *null_rpc_local_sv = -1;
1141
1142         /* This lock corresponds to the CAS from the previous implementation. */
```

```c
        pthread_mutex_lock(&mutex_rpc_done_addr_null);

        /* We wait for the server to be ready. */
        while(global->rpc_done_addr != 0)
            pthread_cond_wait(&cond_rpc_done_addr_null,
                              &mutex_rpc_done_addr_null);

        /* The server is ready, let's execute a RPC. */
        pthread_mutex_lock(&mutex_rpc_done_addr_not_null);
        *null_rpc_global_sv = null_rpc_local_sv;
        pthread_cond_signal(&cond_rpc_done_addr_not_null);
        pthread_mutex_unlock(&mutex_rpc_done_addr_not_null);

        pthread_mutex_unlock(&mutex_rpc_done_addr_null);

        /* We wait until the server is done with our RPC. */
        pthread_mutex_lock(&mutex_rpc_done_positive);
        while (*null_rpc_local_sv < 0)
            pthread_cond_wait(&cond_rpc_done_positive,
                              &mutex_rpc_done_positive);
        pthread_mutex_unlock(&mutex_rpc_done_positive);
    }
#endif
/* ######################################################################### */
/* # End                                                                   # */
/* ######################################################################### */

    if (measurement_location == CLIENTS)
    {
        /* Are we counting cycles? */
        if (measurement_type == NUMBER_OF_CYCLES)
        {
            /* If so, get the current cycle count. */
            end_cycles = PAPI_get_real_cyc();

            if (measurement_unit != TOTAL_CYCLES_MAX)
            {
                /* We return the number of cycles per RPC. */
                iteration_result[client_id + 1] =
                    (double)(end_cycles - start_cycles) /
                    number_of_iterations_per_client;
            }
            else
            {
                iteration_result[client_id + 1] =
                    (double)(end_cycles - start_cycles);
            }
        }
        else
        {
            /* Otherwise, we were counting events. We read the number of
               events. */
            if (PAPI_stop(event_set, values) != PAPI_OK)
                error("PAPI_stop");

            /* We return the number of events. */
            iteration_result[client_id + 1] =
                (double)values[0] / number_of_iterations_per_client;
        }
    }

    /* In the ordered mode, we save the order in which RPCs are serviced. */
    if (execution_mode == SINGLE_RUN_ORDERED)
    {
        order[client_id] = *null_rpc_local_sv;
    }

    for (k = 1; k <= local_number_of_context_variables; k++)
        printf("", (*((volatile uint64_t *)
                     ((uint64_t)context_variables_local_memory_area +
                     k * 2 * sizeof(cache_line_t)))));

    PAPI_unregister_thread();

    return NULL;
}

/* Function executed by the server */
void *server_main(void *thread_arguments_block_pointer)
{
    int i, j, k;
    /* We avoid using global variables in the main loop to limit data cache
       misses. */
    const int number_of_iterations =
        number_of_iterations_per_client * number_of_clients;

    /* Sampling-related variables. */
    const int number_of_samples = number_of_iterations
                                  / number_of_iterations_per_sample;
    const int number_of_iterations_per_sample_m1 =
              number_of_iterations_per_sample - 1;
    int sample_start_cycles, sample_end_cycles;

    const int local_number_of_context_variables = number_of_context_variables;
    const int local_number_of_shared_variables = number_of_shared_variables;

    thread_arguments_block_t thread_arguments_block;
    volatile uint64_t * volatile *null_rpc_global_sv;

    volatile uint64_t *context_variables_global_memory_area,
                      *shared_variables_memory_area;

    /* This variable is used to pin the thread to the right core. */
    cpu_set_t cpuset;

    /* PAPI variables */
    int event_set = PAPI_NULL;
    int events[1];
    long long values[1];
    long long start_cycles = 0, end_cycles;

    /* We pin this thread to the right core. */
    CPU_ZERO(&cpuset);
    CPU_SET(physical_to_virtual_core_id[server_core], &cpuset);

```

```c
    thread_arguments_block =
        *((thread_arguments_block_t *)thread_arguments_block_pointer);

    null_rpc_global_sv = thread_arguments_block.null_rpc_global_sv;

    context_variables_global_memory_area =
        thread_arguments_block.context_variables_global_memory_area;
    shared_variables_memory_area =
        thread_arguments_block.shared_variables_memory_area;

    /* We set the thread's affinity. */
    if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset) < 0)
        error("pthread_setaffinity_np");

    /* Are we monitoring cycles/events on the server? */
    if (measurement_location == SERVER)
    {
        /* If so, we will use PAPI in this thread. */
        PAPI_thread_init((unsigned long (*)(void))pthread_self);

        /* If we are counting events... */
        if (measurement_type == NUMBER_OF_EVENTS)
        {
            events[0] = monitored_event_id;

            /* We initialize the even set. */
            if (PAPI_create_eventset(&event_set) != PAPI_OK)
                error("PAPI_create_eventset");
            if (PAPI_add_events(event_set, events, 1) != PAPI_OK)
                error("PAPI_add_events");

            /* Starting and stopping PAPI once before measurements seems to
               improve accuracy. */
            if (PAPI_start(event_set) != PAPI_OK)
                error("PAPI_start");
            if (PAPI_stop(event_set, values) != PAPI_OK)
                error("PAPI_stop");
        }
    }

    /* We are ready. */
    ready[0] = 1;

    /* We want all the clients to be ready before we start servicing RPCs. */
    for (i = /*1*/0; i <= number_of_clients; i++)
    {
        while (!ready[i]);
            PAUSE;
    }

    /* If the measurements take place on the server... */
    if (measurement_location == SERVER)
    {
        /* ...and if we're county cycles... */
        if (measurement_type == NUMBER_OF_CYCLES)
        {
            /* ...we get the current cycle count. */
            start_cycles = PAPI_get_real_cyc();
        }
        /* Otherwise, we're counting events. */
        else
        {
            /* We start the event counter. */
            if (PAPI_start(event_set) != PAPI_OK)
                error("PAPI_start");
        }
    }

/* ######################################################################### */
/* # Main loop (server)                                                    # */
/* ######################################################################### */
/* First implementation : using spinlocks. */
#ifndef USE_BLOCKING_LOCKS
    /* Sampling is off. */
    if (execution_mode != SINGLE_RUN_SAMPLED)
    {
        /* We start answering RPCs. */
        for (i = 0; i < number_of_iterations; i++)
        {
            /* No address received from a client? That means no RPC has been
               requested. */
            while (!(*null_rpc_global_sv))
                PAUSE;

            if (local_number_of_context_variables > 0)
            {
                for (k = 1; k <= local_number_of_context_variables; k++)
                    (*((volatile uint64_t *)
                      ((uint64_t)(*null_rpc_global_sv) +
                      k * 2 * sizeof(cache_line_t))))++;
            }

            /* We access shared variables */
            for (k = 0; k < local_number_of_shared_variables; k++)
                (*((volatile uint64_t *)
                  ((uint64_t)shared_variables_memory_area +
                  k * 2 * sizeof(cache_line_t))))++;

            /* We notify the client we are done with his RPC by setting the
               variable it provided to i. */
            **null_rpc_global_sv = i;

            /* We are done with our RPC. */
            *null_rpc_global_sv = 0;
        }
    }
    /* Sampling is on. */
    else
    {
        /* Outer loop: one iteration per sample. */
        for (j = 0; j < number_of_samples; j++)
        {
            /* Same as before, except now we get cycle statistics for each
               sample. */
            sample_start_cycles = PAPI_get_real_cyc();
```

```c

            for (i = 0; i < number_of_iterations_per_sample_m1; i++)
            {
                while (!(*null_rpc_global_sv))
                    PAUSE;

                **null_rpc_global_sv = i;
                *null_rpc_global_sv = 0;
            }

            while (!(*null_rpc_global_sv))
                PAUSE;

            **null_rpc_global_sv = i;

            sample_end_cycles = PAPI_get_real_cyc();

            /* We save the results. */
            multiple_samples_results[j] =
                (double)(sample_end_cycles - sample_start_cycles)
                / number_of_iterations_per_sample;
            /* We need to know which core was serviced last. */
            multiple_samples_rpc_done_addrs[j] = *null_rpc_global_sv;

            *null_rpc_global_sv = 0;
        }
    }
/* Second implementation : using blocking locks. */
#else
    /* We start answering RPCs. */
    for (i = 0; i < number_of_iterations; i++)
    {
        /* No address received from a client? That means no RPC has been
           requested. */
        pthread_mutex_lock(&mutex_rpc_done_addr_not_null);
        while (!(*null_rpc_global_sv))
            pthread_cond_wait(&cond_rpc_done_addr_not_null,
                              &mutex_rpc_done_addr_not_null);
        pthread_mutex_unlock(&mutex_rpc_done_addr_not_null);

        /* We notify the client we are done with his RPC by setting the
           variable it provided to i. */
        pthread_mutex_lock(&mutex_rpc_done_positive);
        **null_rpc_global_sv = i;
        pthread_cond_signal(&cond_rpc_done_positive);
        pthread_mutex_unlock(&mutex_rpc_done_positive);

        /* We are done with our RPC. */
        pthread_mutex_lock(&mutex_rpc_done_addr_null);
        *null_rpc_global_sv = 0;
        pthread_cond_signal(&cond_rpc_done_addr_null);
        pthread_mutex_unlock(&mutex_rpc_done_addr_null);
    }
#endif
/* ######################################################################## */
/* # End                                                                  # */
/* ######################################################################## */

    for (k = 0; k < local_number_of_shared_variables; k++)
        printf("", (*((volatile uint64_t *)
                    ((uint64_t)shared_variables_memory_area +
                    k * 2 * sizeof(cache_line_t)))));

    /* We gather statistics if measurements are to take place on the server. */
    if (measurement_location == SERVER)
    {
        /* Are we counting cycles? */
        if (measurement_type == NUMBER_OF_CYCLES)
        {
            /* If so, get the current cycle count. */
            end_cycles = PAPI_get_real_cyc();

            if (measurement_unit != TOTAL_CYCLES_MAX)
            {
                /* We return the number of cycles per RPC. */
                iteration_result[0] =
                    (double)(end_cycles - start_cycles) / number_of_iterations;
            }
            else
            {
                iteration_result[0] = (double)(end_cycles - start_cycles);
            }
        }
        /* We're counting events. */
        else
        {
            /* Otherwise, we were counting events, therefore, we read the number
               of events. */
            if (PAPI_stop(event_set, values) != PAPI_OK)
                error("PAPI_stop");

            /* We return the number of events. */
            iteration_result[0] = (double)values[0] / number_of_iterations;
        }
    }

    PAPI_unregister_thread();

    return NULL;
}

/* Wrapper for malloc that checks for errors. */
void *alloc(int size)
{
    void *result;

    /* We just call malloc and check for errors. */
    if ((result = malloc(size)) == NULL)
        error("malloc");

    /* We return the result. */
    return result;
}

/* This function gets the CPU speed then allocates and fills the
   virtual_to_physical_core_id array. */
void get_cpu_info()
{
    int i = 0;
    FILE *file;
    char text[32];
    float previous_core_frequency = 0, core_frequency = 0;
    int cpu_id, local_core_id, local_number_of_cores;
    int virtual_to_physical_core_id[MAX_NUMBER_OF_CORES];

    /* We use UNIX commands to parse the /proc/cpuinfo file. */
    file = popen("/bin/cat /proc/cpuinfo | "
                 "/bin/egrep \"physical id|core id|cpu cores|cpu MHz\" | "
                 "/bin/sed s/\"core id\t\t: \\|physical id\t: \\|"
                 "cpu cores\t: \\|cpu MHz\t\t: \"//", "r");

    if (file == NULL)
        error("popen");

    /* For each core... */
    while (fgets(text, 32, file) != NULL)
    {
        /* We get the CPU speed. */
        core_frequency = atof(text);

        /* We ensure all cores use the same clock frequency.*/
        if (i > 0 && previous_core_frequency != core_frequency)
            error("all processing cores must use the same clock frequency");

        previous_core_frequency = core_frequency;

        /* We get the CPU ID... */
        fgets(text, 32, file);
        cpu_id = atoi(text);

        /* ...the physical core id on this CPU... */
        fgets(text, 32, file);
        local_core_id = atoi(text);

        /* ...the number of cores on this CPU... */
        fgets(text, 32, file);
        local_number_of_cores = atoi(text);

        /* ...and we use this info to compute the physical core ID (global). */
        virtual_to_physical_core_id[i] = local_core_id +
                                         cpu_id * local_number_of_cores;

        i++;
    }

    /* We update the global cpu_frequency variable. */
    cpu_frequency = core_frequency;

    /* We update the global number_of_cores variable. */
    number_of_cores = i;

    /* We allocate the physical_to_virtual_core_id array. */
    physical_to_virtual_core_id = alloc(number_of_cores * sizeof(int));

    /* We fill the physical_to_virtual_core_id array. */
    for (i = 0; i < number_of_cores; i++)
        physical_to_virtual_core_id[virtual_to_physical_core_id[i]] = i;

    /* We're done with the results of the UNIX commands. */
    if (pclose(file) < 0)
        error("pclose");
}

/* This function writes the usage string to stderr and exits the application. */
void wrong_parameters_error(char *application_name)
{
    const char *usage =
        "usage : %s [-R] [-L] \n"
        "         [-A number_of_runs] [-S #_iterations_per_sample] [-O]\n"
        "         [-s core] [-c #_clients] [-n #_iterations_per_client] "
        "[-d delay] [-r]\n"
        "         [-e event_type] [-m] [-y] [-i]\n";

    /* TODO: print detailed usage. */

    fprintf(stderr, usage, application_name);
    exit(EXIT_FAILURE);
}

/* Error function */
void error(char *reason)
{
    fprintf(stderr, "An error has occured. Cause: %s.\n", reason);
    exit(EXIT_FAILURE);
}
```