Improving the Performance of Java Applications on Multicore Architectures

Julia Lawall julia@diku.dk
Jean-Pierre Lozi jean-pierre@lozi.org
Gilles Muller gilles.muller@lip6.fr
Gaël Thomas gael.thomas@lip6.fr

Abstract

Managed Runtime Environments show poor performance when running legacy Java application on the now common multicore architectures. To overcome this issue, we plan on extending Zeldovich et al.'s [17] hardware parallelization algorithm for event-driven applications to any high-level Java application that uses coarse-grained memory protection. In this document, we review a selection of scientific publications that focus on the issues raised by our approach. We first discuss Zeldovich et al.'s algorithm and its extension proposed by Gaud et al. [8]. We then focus on possible performance optimizations regarding load balancing, cache locality, contention management, IPCs, RPC mechanisms and the handling of heterogeneous architectures. We also briefly discuss the security issues caused by certain performance optimizations.

All along this document, we show the link between each of the chosen works and our project, and we present our reflexions as to how we plan to design our implementation, thereby depicting our current vision of the solution—a vision that will probably evolve during the implementation itself. More information will be available in a forthcoming paper.

1. Context

Multicore hardware is today a reality in all kind of computing architectures ranging from powerful servers to desktop environments and even embedded systems. However, current systems and applications are unable to fully exploit these new architectures: performance is stalling, even though the number of cores is increasing. Taking advantage of multicore hardware is thus one of the most important scientific challenges in the systems domain. Furthermore, addressing this challenge has a crucial economic impact, as highly multicore machines will soon be cheaper and more energy efficient than a cluster of machines with one or only a few cores, but only if the highly multicore machines can be used effectively.

Nevertheless, legacy Java applications are notoriously ill-adapted to multicore architectures. The only concurrency abstraction provided by Java is the synchronized block, which encourages the use of coarse-grained synchronization. Applications cannot be fine tuned for execution on a specific multicore configuration, taking into account, e.g., cache behavior, because such features are hidden by the Java Virtual Machine (JVM). Finally, the training and experience of Java developers is typically more oriented towards aspects of high-level software structuring, and less towards low-level synchronization issues. Nevertheless, legacy Java applications cannot be ignored. The volume of such code and the trend of providing more processing power by increasing the number of cores implies that it is essential to enable legacy Java applications to run efficiently on multicore hardware.

The problem of inadequate support for concurrency has previously been considered in the context of systems software, such as web servers implemented in languages such as C and C++. In this context, the source of inefficiency has been found to be the unpredictability of the operating system process scheduler in its management of concurrency. Because this scheduler is unaware of the pattern of shared data between threads, it may preempt a thread while the thread is holding a lock on shared data, thus effectively holding up all of the other threads that access the data, and it may place threads that share data on different cores, causing cache misses as the data migrates from one core to the other. These problems are exacerbated in the case of coarse-grained parallelism, where the degree of concurrency is already reduced. Coarse-grained parallelism was typically found in the considered web servers, and is typically found in legacy Java programs, for ease of programming.

In the context of C and C++ web servers, the use of an event-based programming model was found to provide a significant performance improvement. This model lifts the problem of scheduling from the operating system to the application level. Specifically, an application is designed as a set of handlers reacting to events. Scheduling of events is entirely controlled by an application-level event engine. This provides predictability, because the event engine controls when, if at all, handler execution is preempted, and flexibility, because the event engine can control the mapping of handlers to cores, to achieve any data locality effect that is desired. Nevertheless, despite the advantages of the event-based programming model, it has so far not seen wide use, due to the difficulty of manually restructuring legacy code to use events.

In the context of this project, we wish to investigate how to improve the performance of legacy Java applications on multicore architectures using the event-based model. In contrast to C and C++ web servers, which run directly on the hardware, Java code runs on top of a virtual machine. Our key observation is then that events can be introduced at the virtual machine level, with no modification to the legacy application. We propose that the virtual machine will recognize the beginning of a synchronized block as the start of an event, and the body of the synchronized block as the event handler. The event engine in the Java Virtual Machine can then ensure that blocks that synchronize on the same value are executed on the same core and without preemption, ensuring that there is no conflict in their access to shared data and increasing the chance that any shared data will be available in the cache when it is required.

The research goal is to develop a Java Virtual Machine incorporating an event engine. The scientific challenge is to design and evaluate various strategies for mapping events to cores and scheduling the associated event handlers. As a result of this project, we hope to improve the performance of legacy Java applications fully automatically on multicore architectures.

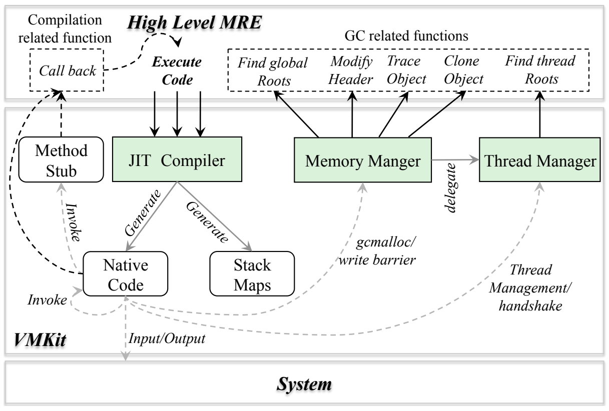

Figure 1: General architecture of VMKit

2. Introduction

In this document, we review the scientific literature which, to our knowledge, tackles best the issues raised by this project. This analysis will allow us to build our solution on the foremost scientific advances in the systems domain. In particular, we review recent works that target the problem of facilitating the parallelization of applications on multicore hardware. Let us note however that even though these works focus on problems similar to ours, their scope is limited to event-driven programs. Our approach is novel in that it treats Java `synchronize` blocks as event handlers, thus effectively extending previous work to any application that uses coarse-grained concurrency management. As for the other works we discuss, even though they provide solutions to individual issues we are faced with, their scope is very different from ours.

Let us now quickly describe the literature we focus on. As we just mentioned, recent publications dealing with the parallelization of event-driven applications are our major source of inspiration since the problems they tackle are very similar to ours. Furthermore, since our approach requires us to execute a considerable number of event handlers—at least one per synchronized block—on several cores, it is necessary to find a way to optimize the remote execution of procedures between cores. Works on various RPC systems (such as URPC [4] and LRPC [10]) and on the migrating thread paradigm provide interesting solutions to this issue. Other works focusing on lower-level optimizations, such as ways to minimize the number of cache misses and stalls, the management of heterogeneous architectures and the implementation of very efficient low-level locks, are also discussed.

Since we plan on adding hardware parallelization capabilities to a Managed Runtime Environment (MRE), the body of literature dedicated to multithreaded operating systems also contains a plethora of information useful to us. Indeed, most of the research focusing on such issues comes from the OS domain: in a way, our project consists in adding OS capabilities to an MRE.

Let us now describe the following sections. In Section 3, we introduce the reader to VMKit, the MRE in which we are going to implement our event-based parallelization algorithm. In Section 4, we discuss recent works dedicated to efficient event-based parallelization techniques. In Section 5, we focus on possible performance optimizations for our implementation. Section 6 deals with security issues. Finally, we conclude in Section 7.

3. VMKit

In this project, we plan on improving the performance of Java legacy applications running above VMKit [9], a generic MRE that has been developed by our team at LIP6. It is worth noting that our approach should be applicable to any MRE: we choose VMKit because it is easily modifiable and because having implemented it provides us with a deep understanding of its inner workings.

The main motivation for the development of VMKit was to allow quickly experimenting with high level languages executing on top of an MRE. Indeed, performing such experiments with Managed Runtime Environments for research purposes is difficult because of their complexity and size (often more than 100,000 lines of code): reimplementing an MRE from scratch is usually out of the question, and modifying one can be a daunting task. To address this issue, VMKit encapsulates most of the functionalities usually found in an MRE. Small and easily-modifiable high-level MREs can be developed on top of it, allowing researchers to experiment with new languages and features without having to implement MRE modules that are irrelevant to their work. There is virtually no limitation regarding the functionalities of the high-level MREs because VMKit does not impose any design decisions (regarding memory allocation, call semantics, etc.). Furthermore, meaningful performance experiments can be performed on VMKit-based projects since VMKit achieves performance comparable to commonly used VMs.

VMKit comprises three main modules: a Just In Time (JIT) Compiler, a Memory Manager and a Thread Manager (as shown in Figure 1). These three modules have not been redeveloped from scratch, instead, VMKit uses state-of-the-art implementations for them, most of VMKit's actual code being the glue between these pre-existing components. More precisely, VMKit uses the LLVM project [1] for its JIT Compiler, the MMTk project [2] for its Memory Manager and the POSIX Threads library to manage threads.

VMKit uses LLVM code as its general-purpose bytecode to communicate with the high-level MREs. A high-level MRE must thus provide a translator from its bytecode to LLVM code. It must also provide memory and thread-related functions to customize its internal design—the facilities provided by VMKit being as generic as possible.

Two high-level MREs have been implemented using VMKit: J3, a Java MRE providing advanced features, and N3, a .NET MRE that was quickly developed but lacks optimizations. In the context of this project, we will have to modify both VMKit and the J3 high-level MRE.

Indeed, VMKit's facilities for handling multithreaded applications are rather poor: hardware parallelism is not supported. This is this case with most, if not all, MREs. We hope to show that our technique is well-suited to handle hardware parallelism and concurrency in MREs and expect that it will be implemented in other MREs in the future.

4. Hardware parallelization of event-driven applications

Zeldovich et al. [17] propose to parallelize the execution of event-driven applications on multiprocessor architectures by executing event handlers in parallel. Our goal is similar, except that we plan to optimize a more general class of applications: any Java application that uses coarse-grained contention management. Indeed, any such application can be turned into an event-driven program by treating each Java `synchronize` block as an event handler[1] and by triggering an event each time such a block must be executed.

To handle contention, Zeldovich et al. allow the user to apply a *color* to each callback, ensuring that two callbacks of the same color are never executed concurrently. This allows for easy-to-handle coarse-grained contention management. We plan on using the same approach, our colors being mapped such that two blocks synchronized on the same resource always share the same color. More complex strategies might be needed for blocks synchronized on several resources or for nested `synchronize` blocks, however.

Zeldovich et al.'s solution is based on the *libasync* library. Experiments with their new library, known as *libasync-smp*, show the efficiency of their approach. They report the results of the parallelization, with libasync-smp, of two examples: (1) a simple web server of their creation that has no particularly processor-intensive operations and (2) the SFS file server whose callbacks are particularly computation-intensive due to encryption. Experiments show that their solution is reasonably efficient in the first case (1.5x speedup on 4 cores,[2]) and very efficient in the second case (2.5x speedup on 4 cores).

Internally, Zeldovich et al. run a single *worker thread* on each core. Indeed, event handlers are supposed to be non-blocking, so only one thread per core is necessary. A task queue that contains a list of all the event handlers (with their arguments) pending for execution is associated with each worker thread. All the callbacks associated with a given color are in the same queue at any given moment, thus preventing any contention problem. Colors are 32 bit unsigned integers; they are mapped to each core via a simple modulo operation: if N cores handle event callbacks, then the color c is executed on core $c \bmod N$ by default. Figure 2 shows the general architecture of libasync-smp.

[1]In this document, we use the terms *event handler* and *callback* interchangeably when talking about event management.

[2]This speedup appears to have been estimated relative to the performance of the modified application running on a single core, not the original application.

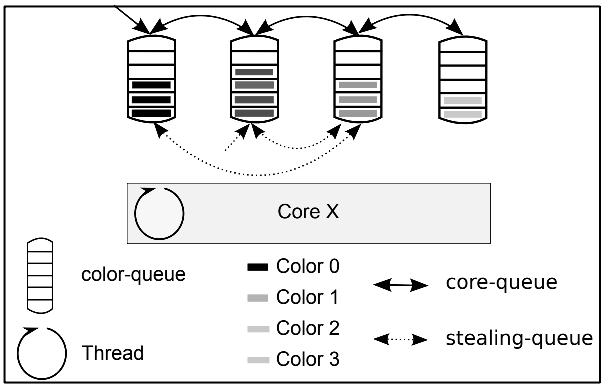

Figure 2: General architecture of libasync-smp

It is worth noting that splitting the execution of a callback in various blocks of various colors can be useful, since callbacks do not necessarily require the same level of isolation during the whole time of their execution. To this end, a `cpucb()` function is provided by libasync-smp. `cpucb()` allows scheduling the execution of a new callback with a new color as soon as the processor is idle, thus effectively making it possible to create a new block using a new color. This splitting mechanism allows for more fine-grained optimizations.

Let us note that, however simple it may be, this mechanism has much weaker semantics than classical concurrency management techniques. Indeed, suppose we wish to allow an unlimited number of callbacks to read data from a shared resource but limit the number of threads accessing this data to one if it is being written. Such semantics cannot be expressed with the callback coloring mechanism. Zeldovich et al. explain that this problem could easily be overcome with minor changes. This issue is of minor importance to us anyways, since we only plan to exploit coarse-grained parallelism with relatively weak semantics.

On a side note, event-based approaches are asynchronous. Calling asynchronous remote functions (i.e. from other threads) can be difficult without specialized libraries. Huang et al. [4] propose a package known as URPC that makes it possible to create efficient, highly customizable RPCs. In particular, they show that their mechanism makes it possible to create efficient asynchronous RPCs. It is worth noting that even though URPC is only meant to be used to experiment with RPC mechanisms, it has been used in OS research projects such as Barrelfish [3].

The naive parallelization of event handlers described in this section can lead to an unbalanced system, in which some cores are given much more work than others. Load balancing techniques are needed to prevent this problem: we will discuss this point, among others, in the next section.

5. Performance optimizations

In the previous section, we described the general idea of Zeldovich et al.'s paper whose algorithm we plan to use in our project. We will however need to fine-tune it to suit our needs, since we will use it in a very different context: instead of providing customizable hardware parallelization capabilities for event-driven programs to developers via a library, we will use it to automatically parallelize any high level program that uses coarse-grained concurrency management. Since we plan to use this algorithm to handle the scheduling of all programs executed within with our virtual machine, performance is a crucial factor. In this section, we look for ways to improve it.

Zeldovich et al.'s algorithm requires load balancing mechanisms to make use of all the available computing power: this is the object of Subsection 5.1. In Subsection 5.2, we focus on cache locality issues. Then, in Subsection 5.3, we focus on the major cause of overhead in multicore architectures : contention. In Subsection 5.4 we will raise the issue of inter-process communication. Subsection 5.5 is dedicated to fast remote execution of code between cores. Finally, in Subsection 5.6, we discuss ways to improve performance on heterogeneous architectures.

5.1. Load balancing

To improve load balancing, Zeldovich et al. propose a basic event stealing approach. We have seen in the previous section that a task queue is associated with each core. When a core is done executing a callback, it looks for another one to execute from its task queue. If the task queue is empty, it attempts to steal work from another core's queue. In order to satisfy the contention invariant (i.e. no two callbacks of a given color are in two different task queues), when a callback is migrated from one task queue to another, all the other callbacks of the same color must be migrated to the new queue. To ensure all the threads of a given color belong to the same queue, libasync-smp uses a 1024 element array that indicates which core should execute all colors which are congruent to $n \pmod{1024}$.

Gaud et al. [8] propose that libasync-smp's work stealing algorithm offers suboptimal performance. They explain that this algorithm ignores three crucial issues. First, they argue that libasync-smp should take the hierarchies of caches and DRAMs into account. These hierarchies have a huge impact on the performance of multicore processors. Interestingly, Zeldovich and al.'s experiments point out that their algorithm shows poor performance on NUMA architectures. It is obvious that stealing work from nearby processors or cores is much more efficient than stealing work from distant cores. Second, Gaud et al. argue that certain tasks should not be stolen since they tend to pollute caches because of the data that they access. Therefore, tasks that access large, long-lived data sets should not be stolen. Third, the execution time of the stolen work should be taken into account. The naive algorithm is extremely inefficient (showing worse performance than the unparallelized libasync library) when the time needed to steal callbacks from a task queue takes longer than executing the callbacks themselves. The execution time of event handlers is thus an important variable that should not be ignored.

Gaud et al. propose an improved algorithm that attempts to solve these issues. Their algorithm is especially good at addressing the first issue: it uses the cache hierarchy provided by the Linux kernel in the `/sys` filesystem to order the list of cores that is traversed each time a worker thread looks for work to steal. Worker threads start looking for callbacks to steal on adjacent cores, only considering distant cores if no stealable work could be found on the closer ones. To solve the second problem, they use a *stealing penalty*, a integer value that must be provided by the developer. A high stealing penalty means that stealing the corresponding job is costly. Finding such a value automatically is obviously very hard, indeed, the causes of cache pollution are diverse and hardware-dependent. Since we aim to improve the performance of legacy applications, we obviously cannot ask the developers to provide us with such a value for each application. To solve the third problem, Gaud et al. need to know the execution time of each callback in order to estimate whether it is worth stealing them or not. Once again, we will not be able to obtain such handcrafted values for legacy applications.

Gaud et al. implemented their algorithm in a library named *Mely* (for Multi-core Event LibrarY). Their performance experiments show major speedups compared to libasync-smp. They reproduced the experiments performed by Zeldovich et al. in their initial paper (albeit with a slightly different web server) and obtained performance gains of +25% compared to libasync-smp without workstealing and up to +73% compared to libasync-smp with workstealing. These results are impressive, but let us keep in mind that they used fine-tuned values for their parameters (i.e. the stealing penalty and estimates of execution times). Finding good values for these parameters automatically is a complex problem that we will address in the next subsection.

5.2. Cache locality

We have seen that, to implement our algorithm efficiently, we will need to (1) estimate the execution time of synchronized blocks, and (2) estimate the cache locality penalties caused by work stealing. Both measurements are correlated since the execution time is highly dependent on the number of cache misses.

Several strategies can be considered to estimate the execution time: for instance, one could simply memorize how long executing a callback takes, then use this value as an estimate for later calls. On the first call, one could use the information gathered about callbacks of the same color to produce an estimate. To our knowledge, finding ways to evaluate the execution time of callbacks has not yet been considered in a context close enough to ours. This subject requires original research, which is outside the scope of this document.

We will instead focus on cache locality issues, about which two recent works have provided us with considerable insight: Pesterev et al.'s DProf [15] and Koufaty et al.'s bias scheduling [13].

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 what type of misses (due to sharing, associativity or capacity overload) are most common for each data type; the *Working Set* view indicates what 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; and the *Data Flow* view shows

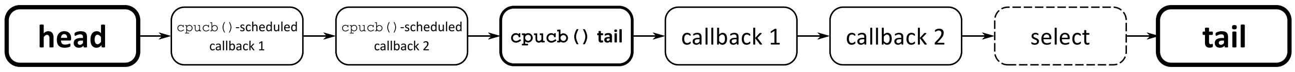

Figure 3: An example task queue from libasync-smp

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 structures 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 (<15%).

DProf is interesting to us at two levels. On a first level, the tool itself will probably be useful when experimenting with work stealing strategies, since it will allow us to monitor information about cache misses, thereby allowing us to compare different implementations and algorithms. On a second level, DProf's implementation itself is extremely interesting, since we will have to use 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 [15] for more information.

Koufaty et al [13] 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. In their implementation, they estimate 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.

5.3. Contention management

Our project raises the issue of contention management at two levels. At a higher level, colors are used to manage contention. At a lower level, the implementation of the color management mechanism has to use locks and other mechanisms to handle concurrency. Since we are looking for ways to optimize our implementation, in this subsection, we of course focus on the lower level. We first discuss strategies to limit the size of critical sections in Subsection 5.4.1. Another way to limit contention is to limit the use of shared data structures. We discuss this point in Subsection 5.4.2. Finally, we focus on the locks themselves, whose efficiency is crucial to any multithreaded environment (Subsection 5.4.3).

5.3.1. Limiting the size of critical sections

A good example of how limiting the size of critical sections improves performance is given by Gaud et al.'s improvements over Zeldovich et al.'s libasync-smp.

With libasync-smp, each core runs a worker thread which uses a *task queue* as its main data structure. When a worker thread is run for the first time, it adds a *select callback* to the end of its queue. Such a callback calls `select()` to detect I/O events[3], enqueues the right callbacks based on which file descriptors have become ready, then puts itself back at the end of the queue. Since the select callback might block the worker thread if no file descriptors are ready, thereby preventing the CPU from executing any other task in its work queue, the select callback uses non-blocking calls to `select()`. If no file descriptors are returned when a select callback is called, a blocking select callback is placed back on the queue instead. The blocking select callback calls `select()` with a non-zero timeout, in all other aspects, it behaves just like the non-blocking select callback. This mechanism, combined with work stealing, guarantees that a worker thread never blocks in `select()` when there are callbacks eligible to be executed in the system.

Each time a worker thread is looking for a callback to execute, it could just take the leftmost one from the queue. This naive approach, however, is not efficient because callbacks working on different data sets would often be executed one after the other, thereby causing a lot of cache misses. To improve data locality, the worker thread explores the whole queue, looking for a callback whose color

[3]The paper is unclear as to how other types of event are handled.

```
core_set = construct_core_set();
foreach(core c in core_set) {
    LOCK(c);                                        (1)
    if(can_be_stolen(c)) {                          (2)
        color = choose_colorsto_steal(c);           (3)
        event_set = construct_event_set(c, color);  (4)
    }
    UNLOCK(c);                                      (5)
    if(!is_empty(callback_set)) {
        LOCK(myself);                               (6)
        migrate(callback_set);                      (7)
        UNLOCK(myself);                             (8)
        exit;
    }
}
```

Figure 4: Pseudo-code of libasync-smp's stealing algorithm, executed by each thread at regular intervals

is the same as the last one that has been run. Indeed, callbacks of the same color are more likely to access the same data sets: grouping their execution allows for better cache locality.

Likewise, callbacks scheduled with `cpucb()` are added to the left of the queue (as shown in Figure 3) in order to increase the performance of chains of `cpucb()` callbacks from the same client request. Indeed, chains of `cpucb()` callbacks coming from the same client share the same state: executing them one after the other decreases the likeliness of cache misses.

The implementation of libasync-smp is inefficient due to contention on the task queues. Indeed, each time a core looks for work to steal from a worker thread, it locks this thread's whole task queue as well as the local task queue. When a task queue is locked, its corresponding worker thread cannot look for callbacks to execute, and other threads cannot try to steal its work.

More precisely, the pseudo-code of libasync-smp's stealing algorithm is shown in Figure 4. This code is executed at regular intervals by each worker thread to try to steal work from other threads. A worker thread starts by identifying the set of cores that handle callbacks. Then, for each core : (1) it locks that core's task queue; (2) it determines whether there is work to steal on that core (this is the case iff there are at least two different colors in the queue). If so, it (3) chooses the color to steal, by scanning the queue, looking for a color that is not currently being run and that is associated with less than half the events in the queue (such a color may not exist). The worker thread then (4) constructs the corresponding set of callbacks, thereby removing them from the core's task queue, and it (5) releases the previously acquired lock. If the set of callbacks is not empty, (6) the worker thread then locks its task queue, adds the events from the even set to its own task queue, and releases the lock.

As we can see in this algorithm, libasync-smp locks the task queues during the execution of time-consuming procedures : `can_be_stolen()`, `choose_color_to_steal()`, `construct_event_set()` and `migrate()` all traverse whole task queues. This can lead to major overheads caused by the locks, especially when the task queues contain a large number of callbacks.

Gaud et al.'s improved implementation (a.k.a. Mely, for Multi-core Event LibrarY), uses much finer-grained locking in order to limit contention. Mely uses one queue per color for each worker thread. These *color queues* are chained together in a doubly linked list. This linked list is called a *core queue*. The architecture of the Mely runtime is shown in Figure 5. With this architecture, when a core needs to choose the next callback to process, it just takes the first callback from the first color queue. Such a naive approach could lead to starvation however, therefore, there is a threshold over the number of callbacks of the same color that can be processed in a row. When a color queue is empty, it is removed from its core queue.

When a worker thread registers an event, it has to find the corresponding core queue. To this end, a small (64 KB), statically allocated array keeps track of mappings between colors and core queues. If the event must be added to a color queue that does not exist yet, the producing thread is responsible for creating the color queue and adding it to the core queue of the corresponding worker thread.

Mely still uses one lock per thread (for its core queue), but the its critical sections are much shorter. Indeed, there is no need to traverse the whole task queue to find all the callbacks of a given color anymore, and migrating all the tasks of a given color from one thread to another does not require copying, a simple operation on pointers being sufficient to migrate a color queue from one core queue to another.

Performance experiments show that, even without the advanced work stealing techniques that we discussed in the previous subsection, Mely outperforms libasync-smp. Gaud et al. used a benchmark called *unbalanced* that implements a fork/join pattern : at each round, 50000 events are registered on the first core. 98% of these events are very short (100 cycles), whereas the other events are much longer (between 10 and 50 Kcycles). All of these events are registered with different colors and can therefore be processed concurrently. When all events have been processed, a new round begins. On this benchmark, libasync-smp with work stealing only manages to process 122 Kevents/s, whereas Mely, with its advanced load balancing techniques disabled, is able to process 1195 Kevents/s. It is worth noting that libasync-smp without work stealing is able to process 1310 Kevents/s (Mely without work stealing processes 1265 Kevents/s), which shows how much poorly managed contention can be detrimental to performance.

Thus, by limiting the size of critical sections, Gaud et al. greatly improved performance. Of course, in our project, we will use their smart management of queues—or at least, base our approach on theirs.

5.3.2. Limiting the sharing of global data structures

Another way of limiting contention is to limit the sharing of global data structures. Wickizer et al. wrote an operating system named Corey [16] that aims to be very efficient on architectures having a large number of cores. Corey integrates the concept of limiting the sharing of global data structures into the very core of the operating system.

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 need update it, unless directed otherwise by the application. This is a very general principle that could probably be beneficial to our project: indeed, threads from the high-level applications are likely to use data structures provided by the MRE, and handling hardware parallelism with a very efficient event-based algorithm is useless if contention over internal data structures is too detrimental to performance.

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 the 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 cores, or a separate address space per core. This can be inefficient if only a subset of the cores 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 allows to lower 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.

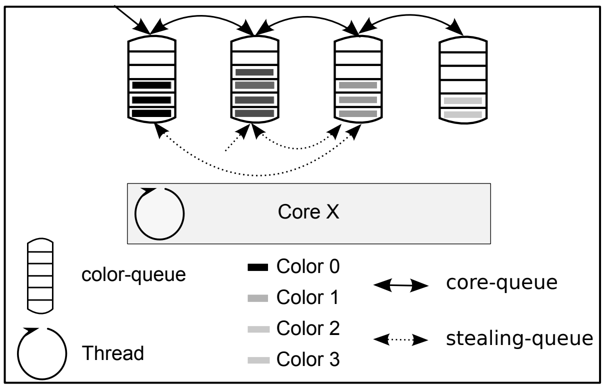

Figure 5: The callback queue structure in Mely

Baumann et al. [3] 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. [7] 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 Subsection 5.4. Before discussing these matters that bring us well beyond traditional contention management mechanisms, let us try to optimize the most common tool in traditional contention management techniques : the lock.

5.3.3. Optimizing locks

Let us now consider contention at a lower level: instead of limiting the size of critical sections or the number of shared structures to improve performance, is it be possible to improve the locks themselves? Both libasync-smp and Mely use spinlocks to manage contention in their implementation, for instance. One could wonder if this is the most efficient approach. Two opposing locking strategies are generally used: to wait for a lock to be released, a thread can either spin, actively polling a resource; or block, 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 is known as *priority inversion* and can drastically impede performance.

Block-based approaches usually fare better 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 true with 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

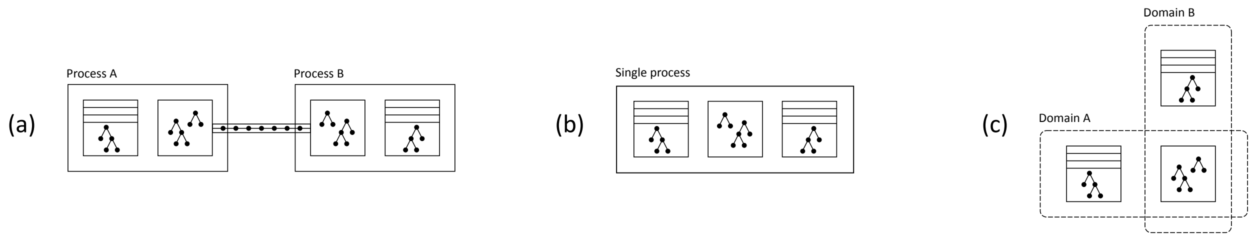

Figure 6: Two components sharing data (a) using standard, marshalled communication over pipes, (b) directly, both components being embedded into the same process, and (c) using Opal's approach with memory protection domains decoupled from execution domains.

have been developed, but they still perform rather poorly, simply offering other tradeoffs between reactivity and CPU usage. Johnson et al. [12] proposed an interesting alternative to spinlocks and blocking locks that we plan to integrate in our solution.

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 clients 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. Since the locking implementation we will choose will be used for all the locks of all high-level applications run on the virtual machine, choosing the most efficient algorithm we can find is crucial: load control is an ideal candidate.

5.4. 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. MREs that handle hardware parallelism are similar in this regard, with the main difference that instead of dealing with kernel threads (i.e. processes), these MREs deals with user threads. Two opposing approaches are 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. [5], 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. [3]), Fähndrich et al. [7, 11]) that, on the contrary, shared memory should be prohibited by the OS. Both approaches have various repercussions on performance, security, and ease of development. In this subsection, we focus on performance only. We will briefly discuss the security issues involved in Section 6. As for the ease of development, this matter is beyond the scope of this document since we will only change the inner workings of an MRE in our project, without having any direct influence on the development of higher-level applications on a semantic level.

5.4.1. 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 security mechanisms. We will focus on security issues in a later section.

The Opal operating system, designed by Chase et al. [5], 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 6, 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 global address space is therefore very efficient. However, it has several drawbacks, in particular, using a single address spaces makes compilation more difficult, which can be a problem in an MRE where the compilation subsystem is already fairly complex, and hard to modify[4].

5.4.2. 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.

[4]With VMKit, it is almost impossible to modify the compilation subsystem without breaking compatibility since it uses LLVM as its JIT compiler.

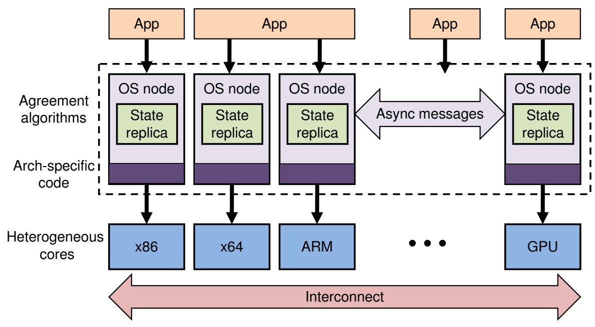

Figure 7: General architecture of Barrelfish

5.4.2.1. Barrelfish

Baumann et al. [3] argue that designing OSes that handle better architectures having a large number of cores[5] 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 8.

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.

Through these experiments, Baumann et al. show that the performance of their messaging facilities is compara-

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

ble to L4's IPCs, which is rather good even though L4 is not a fully-fledged, 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 [7] (and its evolutions, such as Helios [14]) also explored this path: we will focus on this operating system in the next subsection.

5.4.2.2. 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. We will discuss this issue more in detail in Section 6. In the current subsection, we mostly focus on performance issues.

Singularity [7] 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 always belongs to a single thread at any given time. To

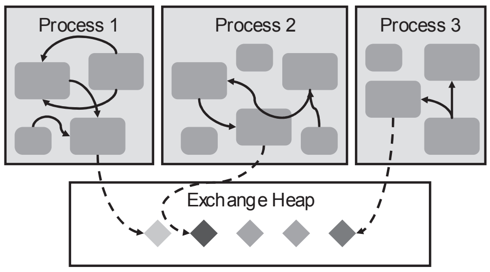

Figure 8: Singularity's exchange heap

this end, data on the GC heap is separated from data in the *exchange heap*—the heap that is used to exchange all data between processes (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 for memory blocks from the exchange heap whose pointers are exchanged *via* messages: a special type of object, *TCell*, is provided to track them explicitly. 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.

Singularity thus offers more proof of the efficiency of message-based communication. We will see in Section 6 that Singularity is also notable for its strong modularity, dependability and safety.

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.

5.5. Fast transfer of control

Optimizing the transfer of data from one thread to another is a good way to improve performance, since it is linked with the major problem of hardware parallel 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 mainly in writing a scheduler that dispatches jobs to cores (and their corresponding worker threads). Given the large number of events that will have to be processed, efficiency is a crucial matter.

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

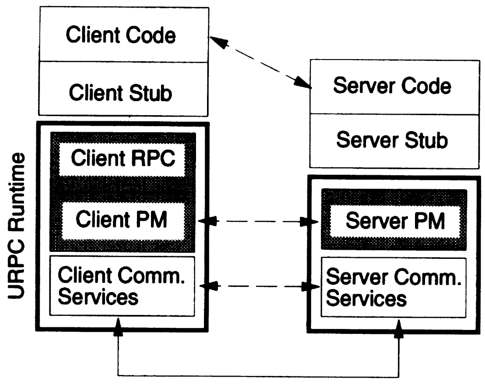

Figure 9: Architecture of the URPC runtime library

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 [4], Huang et al.'s LRPC [10] and Ford et al.'s proposition of a migrating thread model [6].

URPC [4] aims to facilitate the design of highly-specialized RPC systems. Using custom RPC systems instead of generic ones allow to handle 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 9) allows to customize the *protocol machines* of the RPC mechanism (*Client PM* on Figure 9) 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 allows 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 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 [10] (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[6]. 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 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.

Ford et al. propose another compelling way to speed up transfer of control between threads: the *migrating thread model* [6]. In traditional OSes, threads belong to processes and cannot move from one process to another. With the

[6]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.

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.

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 the needed parts of the scheduler code. 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 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.

5.6. 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 event-based scheduling algorithm on heterogeneous architectures could be an interesting long-term goal. This is the object of this subsection.

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*). In addition to the traditional *Clocks Per Instruction* (CPI) metric, we mentioned in Subsection 5.2 that 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 heterogeneous architectures in the general case: these architectures usually have structurally diverse cores. Instead, 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 allows 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.

For instance, we discussed multikernels in Subsection 5.4.2.1. 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 [14], an OS based on Singularity (cf. Subsection 5.4.2.2), 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 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 reminiscent 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 because this subject is beyond the scope of our project for now. Our goal is to implement an efficient event-based hardware parallelization algorithm on standard architectures using regular CPUs. Using NICs, GPUs and other auxiliary cores with multiple ISAs will only be considered in the long term, if our project is successful.

This discussion of possible optimizations in heterogeneous environments ends our section on performance improvements. We will now focus on another issue that is crucial to applicative virtual machines: security.

6. Security

Isolation between components has been a transversal subject throughout this document. Indeed, in multicore computing, adjusting the balance between isolation and sharing is one of the main levers of actions researchers have at their disposal. In the previous section, we reviewed several isolation mechanisms as ways to improve performance: reducing the amount of contention, for instance, can be viewed as increasing isolation. More importantly, the dichotomy between shared memory and message passing that we presented in Subsection 5.4 as a matter of inter-process communication can be viewed as an opposition between weaker and stronger isolation between components.

Isolation issues do not only influence performance, they also influence security. Indeed, less isolated components require stronger mechanisms to ensure security, whereas naturally isolated components are often praised are more secure. Security between processes is important in a virtual machine, on which malicious software that aims to attack other processes running on the top of the virtual machine is a major security issue.

One could wonder why we have not presented the various isolation paradigms as security issues first. The reason is simple: in our project, we are interested in improving the performance of applications on multicore architectures. Therefore, we will base our decision of which isolation mechanisms we will use based on performance, fixing security issues afterwards if needed, rather than the opposite.

In other words, even though modifying the security mechanisms of VMKit might seem to be beyond the scope of our project, we have seen in Section 5 that various choices might lead us to change our isolation model, thereby threatening (or improving) security. This is the object of this section.

More precisely, among the OSes that we presented throughout this document, two unconventional approaches have been considered: range-based memory protection mechanisms and the reject of all shared memory. We discuss these two approaches in the two following subsections.

6.1. Protecting memory areas

We already talked about Opal which overcomes the loss of security caused by its single address space with a traditional capability-based security system. Opal uses three mechanisms to ensure security: *protection domains*, *portals* and *resource groups*.

Protection domains are similar to processes in that they encapsulate an execution context. However, since Opal uses a global address space, protection domains cannot protect their data with virtual address spaces, instead, they own the rights to access a specific set of segments: domains are passive protection contexts within the global virtual address space. Portals are entry points to domains, identified by an integer. Any thread that knows the identifier of a portal can make a system call that transfers control into the domain associated with the portal. Threads entering through a portal begin executing at a global virtual address that is a fixed attribute of the portal, specified by its creator. This allows the creator of a protection domain to control what is executed within its owned memory: such a mechanism improves security since other threads can only manipulate the contents of a protection domain through the facilities provided by the creator of the domain. Resource groups allow threads to access resources: every call to create an Opal resource must pass a capability for a resource group as an argument. Protection domains have a *current resource group* that they can update dynamically.

We mentioned in Subsection 5.3.2 that Corey allows to protect memory by address ranges. In Corey, an address range is an abstraction for a range of virtual-to-physical mappings, allowing to share parts of the address space between cores. Corey only aims to improve performance with this mechanism however, not security.

6.2. Dismissing shared memory altogether

We presented two families of OSes that dismissed shared memory altogether to use message passing instead: multikernels and OSes based on Singularity (Singularity itself, and Helios). Prohibiting shared memory allows for better security mechanisms. Multikernel is conservative in its handling of memory protection. Its protection mechanisms are standard capabilities. However, Singularity's take on security is more innovative.

One of the major goals of Singularity is to allow for stronger specification and cleaner separation between components thanks to the message-passing paradigm. All programs running on the Singularity platform must be written in Sing#, a type-safe, garbage-collected and feature-rich language based on C#.

Furthermore, the OS enforces *channel contracts* over the channels. A contract describes the messages, message argument types and valid message interaction sequences for a channel as a finite state machine. Contracts allow for a cleaner, better specified isolation between components.

In Subsection 5.4.2.2, we described Singularity's strong ownership invariants. We saw that these mechanisms allow Singularity's processes to share their address space (like in Opal) and to dismiss hardware protection altogether (as we have seen in Subsection 5.4.2.2) without decreasing security.

Singularity also uses a *sealed process architecture* [11] in order to improve modularity and security: in addition to completely prohibiting the use of shared memory, Singularity proscribes dynamic code loading and self-modifying code.

More precisely, Fähndrich et al.'s define the sealed process architecture as the combination of four invariants: (1) code within a process does not change once the process starts execution, (2) data within a process cannot be directly accessed by other processes, (3) all communication between processes occurs through explicit mechanisms, with explicit identification of the sender and explicit receiver admission control over incoming communication and (4) the system's kernel API respects invariants (1), (2) and (3).

The first invariant (1) ensures that the code found in applications is the only code that they will ever execute, allowing publishers to issue certificates for their programs. Since the identity of programs can be verified, it can be used to assign access rights to processes. This can be used in combination with user identification to improve security mechanisms. (2) and (3) ensure that all communication is explicit, visible, and controllable by the system (via contracts, as we have seen earlier), and (4) ensures that there are no loopholes allowing processes to execute code dynamically.

In addition to the security guarantees provided by the possibility to digitally sign programs, the open process architecture permits to ensure that extensions and plug-ins will not damage or crash their host program since they run in separate processes and communicate with them through well-defined interfaces.

All of these security mechanisms provided by Singularity are beyond the scope of our project. Indeed, we are only interested in being able to maintain a reasonable security level compatible with our performance optimisations

To conclude, in this section, we have seen that both architectures that use shared memory and those that only rely on message-passing can use reliable security mechanisms. If performance considerations lead us to choose either model, even in a closed context, solutions will be available to overcome security issues.

7. Conclusion

In this document, 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. We started by describing VMKit, the MRE in which we are going to implement our solution, as well as Zeldovich et al.'s algorithm that we plan to use as a basis for our implementation.

We then focused on possible performance optimisations to this algorithm. We discussed Gaud et al.'s work balancing techniques and ways of improving cache locality. We tackled the issue of limiting contention by reducing the size of shared data structures, restricting the sharing of data and improving locks. We then focused on ways to improve IPCs, which lead us to the opposition between shared memory and message-based communication. We explained that finding fast ways to transfer data via IPCs was a efficient approach but that improving the transfer of control was also crucial, which lead us to consider fast local RPC mechanisms. In our last subsection dealing with performance issues, we evoked the possibility of improving performance on heterogeneous architectures.

Finally, in Section 6, we shortly mentioned the security mechanisms used in the various works that we discussed throughout this document.

This review of the scientific literature related to our project provided us with sufficient knowledge to get us started. The next step will be to perform quick experiments on thread migration to ensure that our approach is viable. We then plan to write a simple prototype of our application that we will use to experiment with various performance optimizations. We hope that the results collected in this phase will allow us to implement a fine-tuned version of our algorithm that will prove efficient enough to improve mainstream MREs.

References

[1] LLVM. http://llvm.org.

[2] MMTk. http://jikesrvm.org/MMTk.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Mely: efficient workstealing for multicore event-driven systems. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, 2010.

[9] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–62, 2010.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348, 2010.

[16] 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.

[17] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, pages 239–252, 2003.